Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Lazar

Energijska poraba naprav pri izvajanju programov OpenCL

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Poraba energije postaja vedno bolj pomembna pri razvoju računalniških komponent, pa naj gre za avtonomne sisteme ali pa super-računalnike. Za različne računalniške komponente, na primer procesorje, grafične kartice in koprocesorje, vzpostavite metodologijo za merjenje porabe električne energije. Nato za programe, napisane v programskem jeziku OpenCL, raziščite, kako lahko s konfiguracijo izvajalnega modela vplivamo na porabo.

Rad bi se zahvalil mentorju izr. prof. dr. Urošu Lotriču za strokovno pomoč pri izdelavi in pisanju diplomskega dela. Zahvaljujem se tudi as. Davorju Slugi za omogočanje dostopa do strežnika in pomoč pri rokovanju z njim.

Prav tako se zahvaljujem potrpežljivi in ljubeznivi ženi Maji za nenehne vzpodbude. Hvala tudi mami Martini in tašči Mariji za varstvo otrok.

Nazadnje se zahvaljujem še navihanima prijateljema (J. in D.), ki sta me bodrila in priganjala k delu.

Svoji družini.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPE	central processing unit	centralna procesna enota
DSP	digital signal processor	mikroprocesor za obdelavo signalov
FLOPS	floating-point operations per second	operacije v plavajoči vejici na sekundo
FPGA	field-programmable gate array	programabilni čip
GPE	graphical processing unit	grafična procesna enota
GPGPE	general purpose computing on GPU	splošnonamenska GPE
HPC	high performance computing	visoko zmogljivo računanje
HPL	high performance linpack	knjižnica numeričnih linearnih algoritmov
ICD	installable client drivers	namestitveni gonilniki za odjemalca
MIC	many integrated core	mnogo integriranih jeder
OpenCL	open computing language	odprti računski jezik
PMU	power management unit	napajalna upravljalna enota
PSU	power supply unit	napajalna enota
RAM	random access memory	pomnilnik z naključnim dostopom
SDK	software development kit	paket za razvoj programske opreme

Povzetek

Naslov: Energijska poraba naprav pri izvajanju programov OpenCL

Avtor: Anže Lazar

V diplomskem delu smo raziskali, kaj in v kolikšni meri vpliva na porabo električne energije računalniškega sistema, ki izvaja programsko kodo napisano v programskem jeziku OpenCL. Nekateri taki dejavniki so: čas izvajanja, število delovnih skupin, število zagnanih ščepcev in uporaba lokalnega pomnilnika. Predstavili smo dosedanja prizadevanja na tem področju in trende, ki se razvijajo. Opisali smo tudi programski jezik OpenCL in zakaj se ga uporablja. Predstavili smo sistem, na katerem se izvajajo meritve porabe električne energije, kako se merjenja električne energije nasploh lotimo in katera oprema se uporablja za izvajanje meritev. Predstavljene so tudi težave, vezane na programsko in strojno opremo in kako jih rešimo.

Ključne besede: OpenCL, poraba energije, heterogeni sistemi, visoko zmogljivo računanje.

Abstract

Title: The energy consumption of computer components when executing OpenCL programs

Author: Anže Lazar

We study what and to what extent the power consumption of computer systems is influenced by execution of different programs written in OpenCL programming language. Previous and current trends, that are developing in this field are described. Some identified factors are: running time, number of working groups, number of running kernels and use of local memory. OpenCL programming language and its uses are described. The thesis describes, how do we measure electrical energy in general and what equipment is being used. The computer system on which power measurements were performed is also introduced. Problems related to hardware, equipment and software, that we can encounter, and their solutions are presented as well.

Keywords: OpenCL, energy consumption, heterogeneous systems, high performance computing.

Poglavje 1

Uvod

Merjenje porabe električne energije je že dalj časa aktualno na baterijskih prenosnih napravah (prenosni računalniki, mobilni telefoni, meritvene naprave, medijski predvajalniki in druge). Potrošniki si želimo naprav, ki bi jih lahko z enim polnjenjem uporabljali čim dlje, hkrati pa bi lahko opravljale vedno večje delo. Kapacitete baterij ni mogoče povečevati, ne da bi vplivali na velikost, zato morajo biti naprave tudi bolj učinkovite.

Na drugi strani pa imamo večje sisteme z neprekinjenim napajanjem, kjer poudarka na energetski učinkovitosti ni bilo [22]. Ti sistemi so priklopljeni na električno omrežje, so veliki in zato nepraktični za prenosno uporabo. S tega vidika baterije niso potrebne. To pa ni edini razlog za neučinkovitost teh sistemov. Do neke meje smo lahko povečevali število tranzistorjev na vezjih, saj je vedno manjši proizvodni proces omogočal, da se poraba energije ni kritično povečevala. Zaradi fizikalnih omejitev materialov je to vedno težje dosegati, zato sta varčevanje in optimizacija porabe energije vedno bolj pomembna.

Še posebej se pozna poraba elektrike v podatkovnih centrih in sistemih za visoko zmogljivo računanje. To so večji računalniški sistemi, kot so su-

perračunalniki in gruče. S časom je poraba dosegla tolikšne količine, da lahko skupni stroški lastništva zelo hitro presežejo nakupno vrednost teh sistemov. Zato je jasno, da je iz dneva v dan vedno bolj pomembna poraba energije in z njo tudi učinkovitost. To dokazujeta tudi lestvici Green500 [18] in Top500 [36], kjer so poleg zmogljivosti superračunalnikov prikazane tudi ocenjene vrednosti porabe električne energije.

Heterogeni sistemi so sestavljeni iz različnih vrst naprav. Mednje spadajo centralne procesne enote, grafični procesorji, računski pospeševalniki ali koprocesorji, programabilna vezja FPGA in mikroprocesorji za obdelavo signalov DSP. Vsak tip procesnih enot uporablja drugačno arhitekturo. Pisanje programske kode za take sisteme je bilo nekaj časa zelo zamudno, saj je bilo potrebno napisati program za vsako arhitekturo posebej. Leta 2008 je neodvisen konzorcij za standarde Khronos Group izdal standard OpenCL, ki se še razvija in še vedno rešuje prav ta problem [35].

V tem diplomskem delu nas je zanimala predvsem poraba električne energije programske kode OpenCL glede na izbiro izvajalnega okolja. Zanimalo nas je tudi, ali lahko programer na kakršen koli način zmanjša porabo energije z optimizacijo programov in ali se to splača.

V 2. poglavju smo opisali pregled področja. Raziskali smo, kako se porabo meri, kakšne težave se pojavljajo pri merjenju in izpostavil nekaj del, ki se ukvarjajo s tem področjem. V 3. poglavju smo opisali ogrodje OpenCL, v 4. poglavju opazovan sistem, ki smo ga tekom študija spoznavali preko domačih in seminarskih nalog, v 5. poglavju smo predstavili izvajanje meritev, v 6. poglavju smo predstavili dobljene rezultate in v 7. poglavju zaključke.

Poglavje 2

Pregled področja

Osnovni gradniki procesorjev so tranzistorji. Do neke mere so proizvajalci lahko velikost tranzistorjev zmanjševali (in število le teh povečevali) brez večjega vpliva na porabo električne energije vezja, v kolikor je površina vezja ostajala enaka. Ob zmanjševanju tranzistorjev sorazmerno padata tudi napetost in tok, ki sta potrebna za njihovo delovanje. To opisuje Dennardova teorija skaliranja [3, 22], ki že nekaj let ne velja več. Pri proizvodnem procesu pod 90 nm so dosegli omejitev, ker napetosti in toka ne moremo več nižati, ne da bi vplivali na zanesljivo delovanje tranzistorjev. Rezultat tega je večje gretje vezij in porast porabe električne energije.

Vsak tranzistor za preklop potrebuje moč

$$P = \frac{1}{2} \times C \times U^2 \times f, \tag{2.1}$$

kjer je C kapacitivna obremenitev tranzistorja, f frekvenca preklapljanja in U napajalna napetost [24, 29]. Kot vidimo iz enačbe (2.1), ima največji vpliv na porabo prav napetost, ki je kvadrirana. V zadnjih dvajsetih letih so napetost U najprej zmanjšali s 5 V na 3,5 V in 3,3 V, kasneje pa so uvedli dvonivojsko napajanje. Tako je procesor deloval na 1,6 V, vhodno-izhodne naprave pa na 3,3 V. Med drugim so uvedli še vgrajene regulatorje napetosti

v osnovne plošče, da lahko prilagajajo napetost napravam, ki so priklopljene nanjo. Napajalne napetosti se v času pisanja gibljejo okoli 1 V.

Na porabo energije vpliva tudi povečevanje frekvence delovanja procesnih enot. Procesor Pentium II iz leta 1997 deluje s frekvenco 0,267 GHz, Pentium 4 iz leta 2000 deluje s frekvenco 1,5 GHz in Quad-Core Xeon iz leta 2006 deluje z 2,6 GHz [24]. Sodobni procesorji dosegajo tudi do 5,5 GHz na privzetih nastavitvah. V času pisanja diplomske naloge je svetovni rekord v zviševanju frekvence 8,79433 GHz [15]. Povečevanje frekvence se je v zadnjih letih ustalilo, premostili pa so jo s paralelizacijo. Da bi še dodatno znižali porabo sistemov, so uvedli dinamično prilagajanje frekvence in napetosti (ang. dynamic voltage and frequency scaling) glede na potrebe sistema [14, 29].

Poraba elektrike v večjih računalniških sistemih je z leti neomejeno rasla. Najbolj požrešen sistem s seznama Top500 [36] porabi 19 MW elektrike, za njegovo napajanje pa bi zadostovala na primer hidroelektrarna Dravograd, ki ima najvišjo proizvodno moč 26,2 MW [16]. Povprečje učinkovitosti petindvajsetih najbolj učinkovitih sistemov s seznama Green500 [18] iz leta 2014 znaša 3104 MFLOPS/W, v letu 2015 je naraslo na 3787 MFLOPS/W, v letu 2016 pa je naraslo na 4491 MFLOPS/W [18]. Jasno je, da je iz dneva v dan vedno bolj pomembna poraba energije in učinkovitost naprav. V letu 2016 je prvič po desetih letih povprečna vrednost porabe energije 500 najbolj zmogljivih sistemov padla [18].

2.1 Merjenje porabe električne energije

Raznolikost sistemov za visoko zmogljivo računanje je velika. Izziv predstavlja že način merjenja porabe sistemov, ki vsebujejo številčen nabor različnih komponent.

V članku Power Efficiency in High Performance Computing [22] so avtorji dokazali, da se lahko maksimalni porabi sistemov za visoko zmogljivo računanje zelo približamo z izvajanjem benchmark programov HPL. Benchmark program v angleščini označuje program za primerjanje zmogljivosti več računalnikov. Benchmark HPL je zelo podoben tipičnim računsko zahtevnim znanstvenim izračunom. Čeprav ti testi niso najbolj primerni za primerjanje zmogljivosti različnih računalniških sistemov med seboj [24], ustrezajo našim zahtevam po merjenju najvišje porabe energije. Če bi nas zanimala realna primerjava zmogljivosti, sta bolj primerna benchmarka HPCC (ang. HPC challenge) in SPEC (ang. standard performance evaluation corporation). SPEC je leta 2008 podal iniciativo SPECPower, ki odgovarja na potrebe trga po merjenju porabe energije in zmogljivosti. Postopki in orodja so bolj namenjena strežniški opremi in ne večjim sistemom, kot so sistemi za visoko zmogljivo računanje [32].

Pokazali so še, da lahko predvidimo porabo celotnega sistema glede na meritve, opravljene na enem podsistemu. To je pomembno, ker si lastniki večjih sistemov ne morejo privoščiti izpada delovanja celotnega sistema zaradi izvajanja ponovnih meritev zmogljivosti in porabe. Za tak podsistem lahko vzamemo tudi naš sistem, ki je opisan v 4. poglavju. Meritve so opravljali na dveh sistemih. Na strežniku so jih opravili z merilnikom, ki se ga priključi v stensko vtičnico, na sistemu XT4 pa prek napajalnih vodov v razdelilnih omarah.

Avtorji James H. Laros et al. [25, 26] so se meritev lotili na povsem drug način. Razvili so ogrodje, preko katerega lahko merijo porabo energije s pomočjo senzorjev, že vgrajenih v osnovne plošče strežnikov. To ogrodje jim omogoča zelo natančne meritve s hitrim vzorčenjem (do 100 vzorcev v sekundi). S pomočjo tega ogrodja so ugotovili, da njihov operacijski sistem Catamount Light Weight Kernel (LWK) troši veliko energije v stanju mirovanja. Z odpravo pomanjkljivosti so privarčevali do 80 % energije. Prav tako primerjajo popravljeno različico operacijskega sistema LWK z operacij-

skim sistemom Compute Node Linux (CNL). Pri merjenju porabe aplikacij predstavijo pojem energijski podpis aplikacije. Med izvajanjem posnamejo porabo energije, nato pa lahko s pomočjo že znanih vzorcev določijo, katera aplikacija se izvaja.

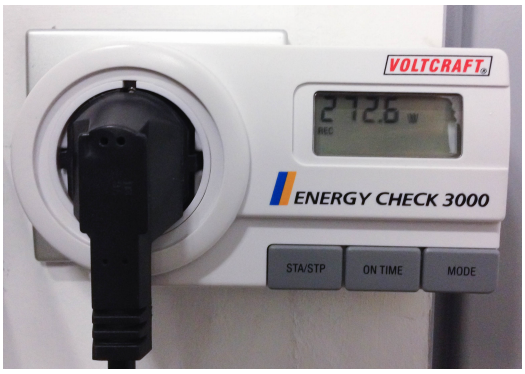
Med samim izvajanjem nočemo, da bi samo izvajanje meritev vplivalo na delovanje sistema (odzivnost, porabo elektrike, gretje in ostalo). Kljub vsem možnim načinom merjenja porabe pa avtorji članka [37] ugotavljajo, da primanjkuje načinov merjenja z visokim vzorčenjem, ki hkrati ne bi vplivali na opazovan sistem. Zato so v članku predstavili ogrodje GreenHPC, ki se vsem tem slabostim izogne, hkrati pa je relativno poceni.

Delovna skupina Energy Efficient High Performance Computing Working Group (EE HPC Working Group) je leta 2012 izvedla raziskavo o uporabi merilnih orodij in prikazu podatkov o učinkovitosti sistemov [11]. Ugotovili so, da ima samo nekaj računskih centrov taka orodja, ki pa so bila opisana kot pomanjkljiva, nedelujoča ali v izgradnji. Zato je delovna skupina EE HPC Working Group izdala smernice o metrikah in podatkih, ki naj bi jih orodja uporabljala in prikazovala. Letos so predstavili posodobljene smernice [11].

Dejstvo je, da obstaja veliko načinov merjenja porabe energije, zato se kakovost meritev zelo razlikuje. V tem kontekstu so najbolj razvite tehnike za merjenje porabe energije na mobilnih napravah. Nekaj teh lahko uporabimo tudi na sistemih za visoko zmogljivo računanje. Porabo energije lahko merimo ali pa vsaj predvidimo na naslednje načine:

- Merilniki, ki jih priključimo med vtičnico električnega omrežja in napajalni kabel naprave (primer je na sliki 2.1), so dokaj natančni in nanje preprosto shranimo meritve. Slabosti so nizka frekvenca vzorčenja, sistem moramo najprej izklopiti iz omrežja in neprimernost izvajanja meritev na gručah, kjer so strežniki napajani preko napajalnih plošč. Lahko pa ga uporabimo na podsistemu gruče, kjer se izpad enega

strežnika ne pozna veliko in tak sistem priklopimo na omrežje preko vtičnice.



Slika 2.1: Energy Check 3000 znamke Voltcraft

- Tokovne klešče za merjenje tokov ponujajo velik razpon vhodnih napetosti in z njimi nimamo težav, ki jih povzroči odklapljanje sistema iz električnega omrežja. Slaba stran je nenatančnost meritev, opravljenih na pletenih žicah [22], zato se takih naprav ne priporoča za resno delo. Primer take naprave je prikazan na sliki 2.2.

Slika 2.2: Voltcraft VC-531 [2]

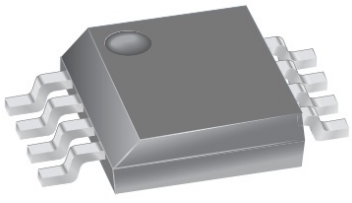
- Samostoječi digitalni merilniki porabe so zelo natančni in podpirajo široko napetostno območje. Večinoma se jih uporablja pri meritvah mobilnih naprav, vendar niso grajeni za merjenje tako visokih tokov, kot jih dosegajo strežniki in sistemi za visoko zmogljivo računanje. Primer takega merilnika je Precision Power Analyzer WT3000E uveljavljenega podjetja Yokogawa [29, 31].

- Vgrajevanje merilnikov na napajanje je dokaj nov pristop, zato takih naprav ni veliko na tržišču. Ponavadi se jih nadzoruje preko interaktivnega ukaznega vmesnika [11, 22].

- Merjenje preko napajalnih vodov v razdelilnih omarah (enotah PDU) je pravzaprav edini način merjenja porabe na celotnem sistemu za visoko zmogljivo računanje. Meritve se opravljajo preko upravljavskega vmesnika zgradbe [11].

- Opravljanje meritev s pomočjo že vgrajenih senzorjev v strojno opremo (na primer osnovne plošče, grafične kartice, napajalnike in koprocesorje) je najbolj enostavno. Primer osnovne plošče je Cray XT [25, 29, 26], grafične kartice imajo skoraj vse že vgrajene bolj ali manj natančne senzorje, prav tako tudi koprocesorji. Programsko poizvedovanje po napetosti in toku z enote PMU (ang. Power management unit) ponavadi omogočajo dodani krmilniki za upravljanje sistema in gonilniki proizvajalca naprave. Poleg gonilnikov ponavadi proizvajalci zagotovijo tudi orodja za dostop do sistemskih podatkov. Lahko uporabimo tudi druga programska orodja, ki pa niso polno združljiva in tako odzivna kot proizvajalčeva [20, 37].

- Hall Effect senzorji za merjenje električnega toka se uporabljajo za integracijo v merilna vezja za merjenje porabe energije. So natančni, hitri in dokaj poceni v primerjavi z ostalimi rešitvami. [31].



Slika 2.3: Senzor ACS712ELCTR-30A-T proizvajalca Allegro [1]

Kjer izvajanje fizičnih meritev ni možno, pa je dobrodošla uporaba simulatorjev, kot sta na primer Wattch [12] in PowerScope [17, 29]. Določena razvojna okolja (na primer Visual Studio podjetja Microsoft) imajo že vgrajena orodja za analizo programske kode, ki glede na profil izbrane naprave predvidi porabo in najbolj neučinkovite dele kode. Profile, ki jih uporablja, sproti popravlja glede na odziv kode v izvajanju. Uporabimo lahko tudi statistično analizo za predvidevanje zagonskih parametrov programov za doseganje najboljše učinkovitosti tako na CPE kot GPE in koprocesorju [27, 33].

Poglavje 3

OpenCL

Odprti računski jezik (OpenCL - ang. Open Computing Language) je nastal iz potrebe podjetja Apple po standardizaciji vmesnikov različnih dobaviteljev. Leta 2008 je neodvisnemu konzorciju za standarde (Khronos Group) predložilo predlog in kmalu za tem s partnerji iz industrije (AMD, IBM, Intel, Nvidia, Qualcomm) ustanovilo skupino Khronos Compute Working Group. Proti koncu leta so izdali prvo različico standarda OpenCL. V skupini je sedaj še veliko več podjetij [9, 35].

Standard definira programski vmesnik (ang. OpenCL API), programski jezik OpenCL C (izpeljan iz jezika ISO C99) in gonilnike raznih proizvajalcev strojne opreme. Programski jezik ima nekaj omejitev, na primer ne pozna rekurzije, kazalcev na funkcije, dinamičnih struktur in polj. Ima pa tudi nekaj že vgrajenih funkcij za delo z nitmi, skupinami procesov in pomnilnikom. Podpira tudi nekatere matematične funkcije in strukture (na primer skalarni in vektorski kazalci). Programi, napisani v jeziku OpenCL C, se imenujejo ščepci (ang. kernels) in so prenosljivi med vsemi viri na sistemu. Ukazi programskega vmesnika se posredujejo gonilnikom naprav, na gostiteljskem sistemu pa so te ukazi le klici funkcij prevedene aplikacije. Ker jih lahko

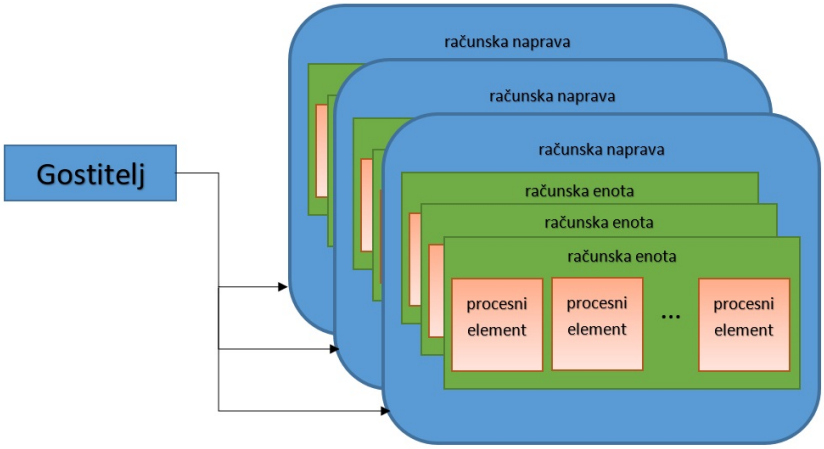
izvajamo sočasno, lahko sistem učinkovito izkoristimo. Uporablja se ga lahko tako na mobilnih napravah kot na superračunalnikih in heterogenih sistemih.

Specifikacija OpenCL [23] ima 4 modele ali koncepte:

- platformni model,
- izvajalni model,
- pomnilniški model in
- programski model

3.1 Platformni model

Model predvideva eno napravo, ki je v vlogi gostitelja, in več računskih naprav. Omogoča abstrakcijo modela strojne opreme za pisanje ščepcev. Računske naprave so naprej razdeljene na računske enote, te pa na procesne elemente (glej sliko 3.1). Kako se abstraktni model poveže na fizični model naprav, določi prevajalnik glede na način optimizacije, ki jo izbere.



Slika 3.1: Platformni model

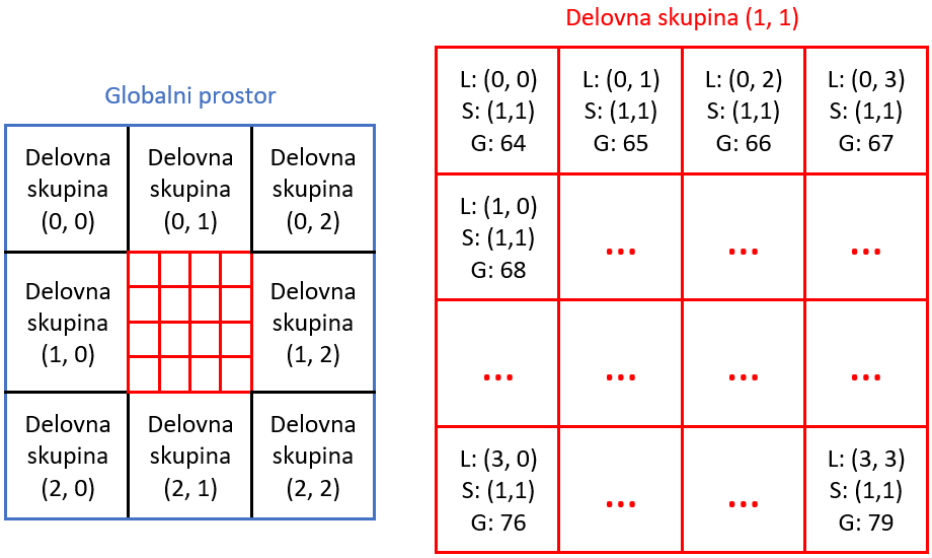
3.2 Izvajalni model

Izvajalni model je definiran preko dveh različnih tipov izvajalnih naprav. Prvi tip je gostitelj, ki izvaja gostiteljski program. Drugi tip pa so računske naprave, ki izvajajo ščepce. Gostitelj je lahko v obeh vlogah. Ščepci so tisti del programa, kjer se izvaja intenzivno računsko delo. To delo je razdeljeno na tako imenovane delavce (ang. work-item), ki pripadajo delovnim skupinam (ang. work-groups).

Ščepec se izvede v naprej definiranem kontekstu. To je okolje za izvajanje ščepcev, upravljanje s pomnilnikom in določanje sinhronizacije med napravami. Kontekst vsebuje eno ali več ukaznih vrst, preko katerih poteka izvajanje operacij. Vsaka naprava ima svojo (ali več) ukazno vrsto, preko katere dobi ukaze. Ukazi v vrsti se lahko izvajajo zaporedno ali v poljubnem vrstnem redu. Slednji se tako izvajajo do sinhronizacijskih točk ali pa do izrecno podanih odvisnih dogodkov.

Če želimo delavcem dodeliti delo, jim moramo dodeliti problemsko področje (preslikati naslovni prostor). To storimo preko N-dimenzionalnega razpona (ang. NDRange). Ta prostor se deli v delovne skupine in je definiran z globalno velikostjo prostora (ang. global size), odmikom v vsaki dimenziji in velikostjo delovne skupine. Delovne skupine so enakih velikosti in morajo pokrivati ves globalni prostor (velikost skupine mora deliti globalno velikost po vsaki dimenziji). Globalni naslov je definiran s koordinatami v eni, dveh ali treh dimenzijah. Vsak delavec je dodeljen delovni skupini in mu je dodeljen lokalni naslov (definiran je enako kot globalni naslov) znotraj te skupine. Tako organizacijo v dveh dimenzijah prikazuje slika 3.2.

Vsak delavec ima torej globalni naslov, naslov delovne skupine, lokalni naslov znotraj skupine (ang. global ID, work-group ID, local ID) in se izvaja na procesni enoti (ang. streaming processor).

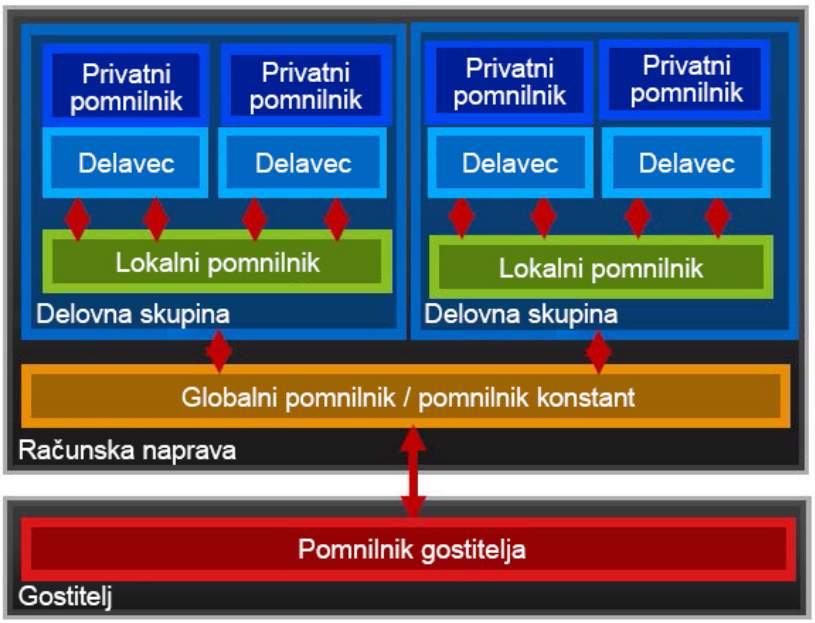


Slika 3.2: Globalni prostor je razdeljen na delovne skupine. Delovne skupine se naprej delijo na delavce, kjer ima vsak delavec svoj lokalni naslov (L), naslov delovne skupine (S) in globalni naslov (G).

3.3 Model pomnilnika

Delavci (ang. workitem) imajo privatni pomnilnik (ang. private memory), do katerega lahko dostopa samo tisti delavec, ki si ga lasti. Znotraj delovne skupine (ang. workgroup) je definiran lokalni pomnilnik (ang. local memory), do katerega lahko dostopajo delavci znotraj iste skupine. Podatke si lahko izmenjujejo preko globalnega pomnilnika (ang. global memory), skupen pa jim je tudi pomnilnik konstant (ang. constant memory). Računska naprava si podatke z gostiteljem izmenjuje preko globalnega pomnilnika, gostitelj pa ima tudi svoj pomnilnik (ang. host memory). Opisana organizacija je prikazana na sliki 3.3.

Pri taki organizaciji pomnilnika se moramo vprašati, kako je s konsistentnostjo podatkov v pomnilniku. Za posamezno nit so podatki konsistentni v privatnem pomnilniku. Na nivoju delovnih skupin so podatki konsistentni samo še ob sinhronizacijskih točkah. Ko se ščepci izvajajo, konsistentno-



Slika 3.3: Organizacija pomnilnika v OpenCL [23]

sti podatkov v glavnem pomnilniku med različnimi skupinami ne moremo zagotoviti. Sinhroniziramo jih lahko, ko se ščepci zaključijo. Podatki so torej konsistentni samo znotraj delovne skupine. Če želimo, da imajo ukazi v ukazni vrsti konsistentne podatke, mora to programer zagotoviti z ročno sinhronizacijo.

3.4 Programski model

Programski model vsebuje kontekst, programsko kodo (ki je lahko izvorna ali pa binarna), seznam ciljnih naprav in navodila prevajalniku za vsako ciljno napravo. Prevajalnik nato prevede izvorno kodo in jo poda v izvajanje (ang. just in time) ali pa uporabi binarno kodo in jo izvede. Prevajalnik OpenCL mora za vsako napravo zagotoviti proizvajalec strojne opreme in gonilnika.

3.5 Opis programske kode

Programska koda sestoji iz dveh delov. Prvi del je programska koda, ki se izvaja na gostitelju in drugi del je programska koda (imenovana ščepec), ki se izvaja na računskih napravah.

Prvi del programske kode smo napisali v jeziku C++. Na začetek programa smo vključili zaglavja knjižnic, ki jih program uporablja. Poleg standardnih knjižnic smo vključili tudi zaglavje knjižnice `CL/cl.h`. Vključitvam knjižnic sledijo definicije konstant. Nekatere definicije konstant niso nujno potrebne, a smo jih definirali zaradi hitrejšega prilagajanja programa za prevajanje. To so konstante za velikost problema, velikost delovnih skupin in indeksa za izbiro naprave in platforme. Ker smo kodo pisali in testirali na operacijskem sistemu Windows in ker smo izvajali meritve na operacijskem sistemu CentOS, smo definirali tudi makro za merjenje časa. Vsak program potrebuje metodo `main`, ki se samodejno zažene ob zagonu. Pred to metodo so po navadi v črkovnem nizu zapisani ščepci, ki pa se v večjih programih preberejo iz samostojne datoteke. Sledi priprava podatkov, ki jih želimo posredovati ščepcu v obdelavo. V pomnilniku smo rezervirali prostor in ga napolnili s podatki

Nato smo vzpostavili izvajalno okolje OpenCL. Kazalce na platforme smo pridobili z ukazom `clGetPlatformIDs` (slika 3.4, vrstica 53). Z ukazom `clGetDeviceIDs` smo pridobili podatke o vseh napravah na izbrani platformi. Znotraj izvajalnega okolja smo definirali kontekst in ukazno vrsto (slika 3.4, vrstici 57 in 58). V kontekst smo vključili platformo in izbrane naprave. V ukazno vrsto pa smo podali kontekst, napravo in nastavitev za zaporedni način izvajanja ukazov.

Delo smo razdelili na podprobleme z določitvijo števila delavcev v delovni skupini in števila vseh delavcev (slika 3.4, vrstici 59 in 60). Pripravili smo prej prebrani ščepec in ga prevedli. Priprava podatkov na napravi poteka s

```
50 cl_platform_id  platform_id[10];
51 cl_uint         ret_num_platforms;
52 ret = clGetPlatformIDs(10, platform_id, &ret_num_platforms);
53
54 cl_device_id    device_id[10];
55 cl_uint         ret_num_devices;
56 ret = clGetDeviceIDs(platform_id[platform_sel], CL_DEVICE_TYPE_ALL, 10, device_id, &ret_num_devices);
57 cl_context context = clCreateContext(NULL, 1, &device_id[device_sel], NULL, NULL, &ret);
58 cl_command_queue command_queue = clCreateCommandQueue(context, device_id[device_sel], 0, &ret);
59 size_t local_item_size = WORKGROUP_SIZE;
60 size_t global_item_size = ceil(n/(float)local_item_size)*local_item_size;;
61 cl_program program = clCreateProgramWithSource(context, 1, (const char **)&source_str, NULL, &ret);
62 ret = clBuildProgram(program, 1, &device_id[device_sel], NULL, NULL, NULL);
63 cl_kernel kernel = clCreateKernel(program, KERNEL_NAME, &ret);
64 cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, vectorSize*sizeof(int), A,
   &ret);
65 ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
66 ret |= clSetKernelArg(kernel, 1, sizeof(cl_int), (void *)&vectorSize);
67 // Start ob pritisku na tipko enter
68 printf("Pres ENTER to start\n");
69 fflush(stdout); scanf("%c", &ch);
70 double t1 = dtime();
71 // Zagon scepca
72 ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_item_size, &local_item_size, 0, NULL,
   NULL);
73
74 double t2 = dtime();
```

Slika 3.4: Del kode lastnih programov, napisanih v programskem jeziku C++

kreiranjem vmesnega pomnilnika (ang. buffer), ki povezuje pomnilnik na gostiteljski in izvajalni napravi, ter nastavitvijo argumentov ščepcu (slika 3.4, vrstice 64-66). Zagon ščepca, merjenje izvajalnega časa in sprostitev pomnilnika so prikazani na sliki 3.4 v vrsticah 68-74.

Drugi del programske kode je ščepec. Označba `__kernel` označuje, da gre za OpenCL ščepec. Oznaka `__global` označuje, da se kazalec nanaša na globalni pomnilnik naprave. Metoda ne vrača nobene vrednosti (oznaka `void`). V primeru na sliki 3.5 vsak delavec preveri, če se nahaja znotraj problemske domene in izpiše število, ki je podano v vektorju `A` na njegovem naslovu.

```
1 __kernel void scepec(__global const int *A, int size)
2 {
3     int i = get_global_id(0);
4     if (i < size) {
5         printf(A[i]);
6     }
7 }
```

Slika 3.5: Ščepec, napisan v programskem jeziku OpenCL

Poglavje 4

Opis opazovanega sistema

Meritve smo izvajali na laboratorijskem strežniku Gpufarm (na sliki 4.1). Strežnik uporablja osnovno ploščo Supermicro X9DRG-QF [34], poganjata

Slika 4.1: Laboratorijski strežnik brez koprocesorja Xeon Phi in grafičnih kartic Tesla

ga dva procesorja Intel Xeon E5-2620, za napajanje skrbita dva napajalnika in za hlajenje uporablja nekaj več kot 10 ventilatorjev. Ima 64 GB sistemskega

pomnilnika, dve grafični kartici Nvidia Tesla K20m in koprocesor Intel Xeon Phi 5110P.

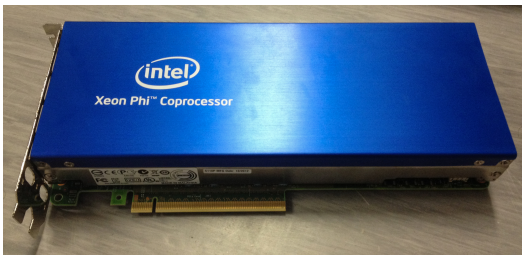
CPE Intel Xeon E5-2620 (verzija 1) [21] je narejen v arhitekturi z razvojnim imenom Sandy bridge. Vsebuje 6 fizičnih jeder in s pomočjo tehnologije hyper-threading lahko poganja 12 niti (ima 12 navideznih jeder). Njegova frekvenca delovanja je 2 GHz in se ob večjih obremenitvah s pomočjo tehnologije Turbo Boost poveča na 2,5 GHz. Njegova teoretična zmogljivost pri enojni natančnosti plavajoče vejice in osnovni frekvenci delovanja je 96 GFLOPS.

Grafična kartica Nvidia Tesla K20m [28] spada tudi med naprave GPGPE. Naprave GPGPE so grafične kartice, ki niso namenjene samo preračunavanju grafike in prikazu slike, ampak tudi drugim izračunom, ki jih je običajno opravljala CPE [5]. Grafična kartica ima 2496 jeder CUDA (ang. Compute Unified Device Architecture), ki delujejo v frekvenčnem razponu 706 MHz - 758 MHz. Vsaka kartica ima tudi 8 GB pomnilnika. Teoretična zmogljivost ene kartice v enojni natančnosti plavajoče vejice je 3524 GFLOPS, v dvojni natančnosti plavajoče vejice pa 1174 GFLOPS. Opazovani kartici (na sliki 4.2) sta nastavljeni na računski način izvajanja (ang. compute mode), torej ne izvajata operacij za prikaz slike.

Slika 4.2: Grafična kartica Nvidia Tesla K20m

Koprocesor Xeon Phi 5110P [20] je zgrajen na arhitekturi Intel MIC

(Many Integrated Core). Od GPGPE se razlikuje po manjšem številu zmogljivejših jeder (grafična kartica Tesla ima 2496 jeder, koprocesor Xeon Phi pa 60). Ker je arhitektura nekoliko bolj podobna CPE, bolje podpira vejitve programov in jo lahko programiramo s standardnimi programskimi jeziki (C, C++ in FORTRAN) [19]. Koprocesor Xeon Phi 5110P ima 60 fizičnih jeder in s pomočjo tehnologije hyper-threading tudi 240 navideznih jeder. Vsako fizično jedro vsebuje zmogljivo 512 bitno vektorsko procesno enoto, ki podpira vse osnovne matematične operacije in sočasno seštevanje z množenjem (ang. fused multiply-add - FMA). Skupaj ima tudi 30 MB predpomnilnika. Vsako jedro ima na L2 nivoju po 512 kB (za podatke in ukaze) in na L1 nivoju po 64 kB (32 kB za podatke in 32 kB za ukaze) predpomnilnika. Teoretična zmogljivost v enojni natančnosti plavajoče vejice je 2021 GFLOPS, pri dvojni natančnosti plavajoče vejice pa 1011,88 GFLOPS.



Slika 4.3: Koprocesor Intel Xeon Phi 5110P

Programska oprema

Na strežnik je naložen operacijski sistem CentOS (ang. Community Enterprise Operating System) verzija 6.8. To je odprtokodna prostodostopna distribucija Linuxa, ki strmi k 100 % kompatibilnosti programskih paketov s sistemom. Arhivi programske opreme za nameščanje so v obliki paketov RPM. Orodje za nalaganje programov in posodabljanje sistema CentOS je osnovano na sistemu yum.

Za namestitev nekaj programskih paketov (`gcc`, `gcc-c++`, `openmpi-devel` in `make`) smo uporabili ukaz `sudo yum install gcc gcc-c++ make openmpi-devel`.

Če vemo, kateri ukaz potrebujemo (na primer `mpicc`), ni pa nameščen noben programski paket s tem ukazom, lahko preverimo, v katerih paketih je prisoten, z ukazom `sudo yum provides */mpicc`. Za priklic pomoči lahko uporabimo ukaz `man <ukaz>` ali pa ukazu dodamo stikalo `-h`.

Namestitev Intelovih knjižnic

Najprej smo morali namestiti Intelove knjižnice in gonilnike za koprocesor Xeon Phi. Potrebovali smo programske pakete Intel Parallel Studio XE, Intel Math Kernel (IMK) in Intel MPI (IMPI). Študenti lahko programsko opremo dobimo zastonj na naslovu: `https://software.intel.com/en-us/qualify-for-free-software/student`. Morali smo se le registrirati in že smo lahko prenesli vse potrebno za delo. Preneseno datoteko smo raztegnili z ukazom `tar xaf <ime datoteke>` in v njej zagnali namestitveno datoteko z ukazom `./install.sh`. Podrobna navodila za namestitev smo dobili na elektronski naslov, ki smo ga podali ob registraciji, lahko pa jih najdemo tudi na spletu. Programski paket smo namestili na privzeto lokacijo `/opt/intel/`. Ker je matematična knjižnica Intel Math Kernel že vsebovana v tem paketu, smo namestiti samo še knjižnico Intel MPI. Knjižnico smo namestili podobno kot prejšnji programski paket na privzeto mesto `/opt/intel/impi/`. V sistemu je bilo potrebno dopolniti okoljske spremenljivke. To smo storili tako, da smo dodali vrstici na sliki 4.4 v datoteko `$HOME/.bash_profile`.

```
export PATH=/opt/intel/impi_latest/bin64/:/opt/intel/
composer_xe_2015.3.187/bin/intel64/:$PATH

export LD_LIBRARY_PATH=/opt/intel/impi_latest/lib64/:/opt/intel
/composer_xe_2015.3.187/mkl/lib/intel64/:$LD_LIBRARY_PATH
```

Slika 4.4: Dopolnitev okoljskih spremenljivk za okolje Intel

Za namestitev gonilnikov smo potrebovali s strani `https://software.intel.com/en-us/mic-developer` prenesti namestitveno datoteko. Navodila se nahajajo v datoteki `Readme`, pomagali pa smo si tudi z vodičem iz knjige [30]. V našem primeru smo gonilnike samo posodobili.

Nastavitev okolja za GPE

Namestitev gonilnikov za grafični kartici Tesla, zbirk orodij CUDA (ang. CUDA toolkit) in prilagojene implementacije benchmarka HPL lahko začnemo po navodilih na naslovu: `http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/`. Ker je bil naš sistem že nastavljen, smo samo preverili, če je vse, kar navodila narekujejo, izpolnjeno. Dodali smo nekaj vrstic s slike 4.5 v datoteko `$HOME/.bash_profile` za razširitev okoljskih spremenljivk.

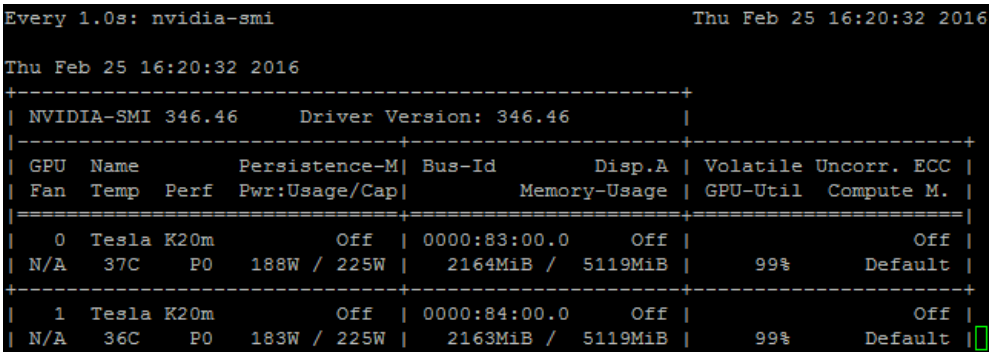
```
# CUDA Settings
export CUDA_HOME=/usr/local/cuda
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64
export PATH=${CUDA_HOME}/bin:${PATH}
```

Slika 4.5: Dopolnitev okoljskih spremenljivk za okolje CUDA

Nazadnje smo testirali pravilno namestitev gonilnika z izvedbo ukaza `nvidia-smi` (primer izpisa je prikazan na sliki 4.6). Izpisati se morajo podatki o vseh nameščenih grafičnih karticah.

Nastavitev okolja OpenCL

Za nastavitev okolja smo potrebovali knjižnico `libOpenCL.so`, ki poskrbi za nalaganje gonilnikov ICD (ang. Installable Client Drivers) nameščenih naprav. V mapo `/etc/OpenCL/vendors/` smo morali dodati vse ICDje (datoteke `*.icd`), ki vsebujejo poti do implementacij OpenCL za določeno napravo. Nazadnje smo potrebovali še vse implementacije, ki jih lahko pre-



Slika 4.6: Primer izpisa ukaza `nvidia-smi` na našem sistemu

nesemo s spletnih strani proizvajalcev. Skupaj z gonilniki smo že namestili paketa za razvoj programske opreme (SDK) proizvajalcev Intel in Nvidia, tako da nam samostojnih SDKjev ni potrebno prenašati. Preveriti smo morali še, če so datoteke ICD nameščene v mapi.

Preko programskega vmesnika OpenCL smo poizvedeli po nekaterih lastnostih strežnika. Naš sistem vsebuje dve platformi, saj so računske naprave od dveh različnih proizvajalcev (prikazano na sliki 4.7). Vsaka platforma vsebuje po dve napravi.

```
platforma[0]:
  NVIDIA CUDA
  naprava[0]:
    DEVICE_NAME = Tesla K20m
    DEVICE_VENDOR = NVIDIA Corporation
    DEVICE_MAX_COMPUTE_UNITS = 13
    DEVICE_MAX_CLOCK_FREQUENCY = 705
    DEVICE_GLOBAL_MEM_SIZE = 5368512512
    DEVICE_MAX_WORK_GROUP_SIZE = 1024
  naprava[1]:
    ima enake lastnosti kot naprava[0]
platforma[1]:
  Intel(R) OpenCL
  naprava[0]:
    DEVICE_NAME = Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
    DEVICE_VENDOR = Intel(R) Corporation
    DEVICE_MAX_COMPUTE_UNITS = 24
    DEVICE_MAX_CLOCK_FREQUENCY = 2000
    DEVICE_GLOBAL_MEM_SIZE = 67521273856
    DEVICE_MAX_WORK_GROUP_SIZE = 8192
  naprava[1]:
    DEVICE_NAME = Intel(R) Many Integrated Core Acceleration Card
    DEVICE_VENDOR = Intel(R) Corporation
    DEVICE_MAX_COMPUTE_UNITS = 236
    DEVICE_MAX_CLOCK_FREQUENCY = 1052
    DEVICE_GLOBAL_MEM_SIZE = 6053646336
    DEVICE_MAX_WORK_GROUP_SIZE = 8192
```

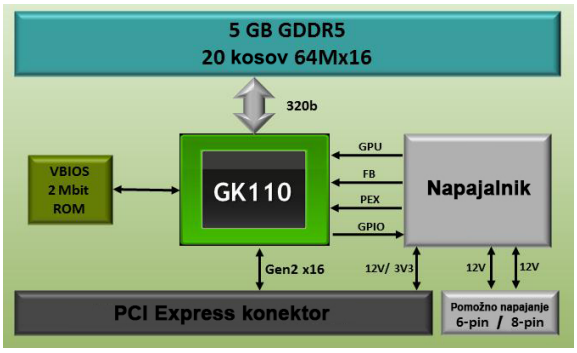
Slika 4.7: Izpis nekaterih lastnosti naprav na našem strežniku glede na pripadajočo platformo

Poglavje 5

Izvajanje meritev

Meritve na sistemu smo izvajali na dva načina. Grafični kartici Tesla in koprocesor Xeon Phi imajo senzorje vgrajene na vezje, zato smo jih lahko izrabili za poizvedovanje po porabi električne energije preko programskega vmesnika. Ker procesor in osnovna plošča podobnih senzorjev nimata, smo primorani porabo CPE meriti z merilnikom porabe. V 2. poglavju smo ugotovili, da tokovne klešče niso najbolj natančne za opravljanje teh meritev, zato smo se odločili za merjenje s stenskim merilnikom Energy Check 3000 proizvajalca Voltcraft (prikazanim na sliki 2.1). Merilno območje te naprave je od 1,5 W do 3 kW s toleranco $\pm$ 2 % $\pm$ 2 W za meritve do 2,5 kW in $\pm$ 4 % $\pm$ 4 W za meritve nad 2,5 kW . Naš sistem ima dva procesorja, zato potrebuje tudi dva napajalnika. Tako smo oba napajalnika preko električnega razdelilnika priključili na stenski merilnik. Meritve smo opravljani v različnih sestavah z in brez grafičnih kartic Tesla in kopreocesorja Xeon Phi.

Porabo grafičnih kartic Tesla smo beležili s pomočjo orodja `nvidia-smi`. To nam omogoča tudi posebna arhitektura, ki jo prikazuje shema kartice na sliki 5.1. Gonilnik lahko preko vodila direktno zahteva podatke o delovanju grafične kartice in o napajanju [13].



Slika 5.1: Shema grafične kartice Tesla K20m [28]

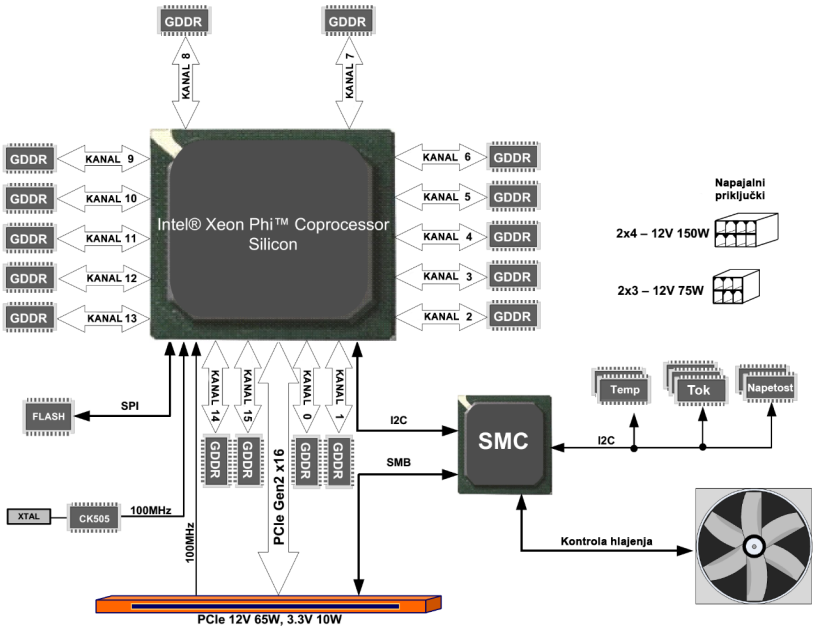
Napisali smo skripo (slika 5.2), ki izkoristi program `nvidia-smi` in je napisana za okolje Linux. Programu poda argumente, ki določajo, katere podatke želimo beležiti. Odločili smo se, da naj nam beleži čas, temperaturo, energetsko stanje, podatke o pomnilniku, napajanju in podatke o frekvenci ure. Te podatke privzeto beleži v datoteko `gpu.csv` s frekvenco 100 Hz.

```
#!/bin/bash
filename="gpu"  # privzeto ime
if [ $# -gt 0 ]; then
  filename=$1
fi
# zazenemo belezenje GPE naprav
nvidia-smi --query-gpu=timestamp,index,name,temperature.gpu,pstate,utilization.gpu,utilization.memory,memory.free,
memory.used,memory.total,power.draw,clocks.gr,clocks.sm,clocks.mem --format=csv --loop-ms=10 --filename=$filename.
csv &

pid=$!
echo $pid
trap "kill $pid 2> /dev/null" EXIT

# zazenemo program
mpirun -n 2 ./run_linpack &
pid2=$!
echo $pid2
trap "kill $pid2 2> /dev/null" EXIT
while kill -0 $pid2 2> /dev/null; do
    sleep 2
done
sleep 1
kill $pid
# Disable the trap on a normal exit.  nvidia-smi
trap - EXIT
```

Slika 5.2: Skripta za izvajanje meritev na GPE

Ko so Intelovi inženirji zasnovali arhitekturo Intel MIC, so mislili tudi na energetsko učinkovitost naprav, saj se zavedajo, da je učinkovitost glavno vodilo pri zasnovi sistemov za visoko zmogljivo računanje. To dokazuje tudi shema vezja, ki je prikazana na sliki 5.3. Na shemi opazimo sistemski krmilnik

(ang. System Management Controller - SMC), ki je zmožen beležiti podatke o napravi, tudi ko naprava miruje. Novejši Intelovi procesorji družine Xeon lahko na enostaven način sami berejo podatke o napravi, nekoliko starejšim pa jih lahko zagotovi sistemski krmilnik preko posebnega protokola [4].



Slika 5.3: Shema koprocesorja Xeon Phi [4]

Na koprocesorju Xeon Phi lahko porabo električne energije merimo na nekaj različnih načinov [4, 20]. Odločili smo se za orodje `micsmc` iz zbirke *Intel Manycore Platform Software Stack* (Intel MPSS) [7]. Če program zaženemo preko ukazne vrstice s pomočjo istoimenskega ukaza in brez argumentov, se nam preko okenskega sistema X (X11) prikaže grafični vmesnik. Za potrebe programskega dostopa do podatkov, pa lahko s podajanjem argumentov, podatke pridobimo direktno v svoj program ali skripto.

Napisali smo skripto (slika 5.4), ki poizveduje po podatkih koprocesorja Xeon Phi, dokler se program v izvajanju ne zaključi. S stikali `-t` `-f` `-m` in `-c` določimo podatke, ki jih želimo pridobiti. To so podatki o temperaturi, frekvenci, porabi energije, pomnilniku in stanju jeder.

```
#!/bin/bash
filename="mic.log"  # privzeto ime
if [ $# -gt 0 ]; then
  filename=$1.log
fi

echo "Start time: $(date +"%T")" | tee -a $filename
while true; do
    time=$(date +"%T")
    echo "Time: $time $(micsmc -t -f -m -c)" >> $filename
    sleep 0.01
done
echo "End time: $(date +"%T")" | tee -a $filename
# Disable the trap on a normal exit.
trap - EXIT
```

Slika 5.4: Skripta za izvajanje meritev na koprocesorju Xeon Phi

5.1 Testno breme

Za ugotavljanje najvišje porabe sistema v različnih sestavah smo uporabili benchmark program High-Performance Linpack (HPL). Za izvajanje testov na GPE smo morali prenesti in prevesti posebno implementacijo, ki je optimizirana za arhitekturo CUDA. Za izvajanje na Intelovih napravah pa smo lahko uporabili Intelovo optimizirano implementacijo ali pa odprtokodno.

Za primerjavo smo uporabili zbirko benchmark programov SHOC (ang. The Scalable HeterOgeneous Computing benchmark suite) [10]. Namenjen je testiranju zmogljivosti in zanesljivosti heterogenih sistemov za visoko zmogljivo računanje. Osredotoča se na sisteme, ki uporabljajo GPE in več-jedrne procesorje ter podpirajo standard OpenCL. Lahko se ga uporablja na gručah ali posameznih strežnikih.

Napisali smo tudi programe za vektorsko seštevanje (podpoglavje 5.1.3), množenje vektorjev po komponentah (podpoglavje 5.1.4), skalarno množenje (podpoglavje 5.1.5) in izračunavanje histograma (podpoglavje 5.1.6). Nato smo izmerili njihove porabe in jih primerjali med seboj.

5.1.1 High-Performance Linpack

Benchmark HPL je program, ki rešuje naključen linearen sistem enačb v dvojni natančnosti aritmetičnih operacij na sistemih s porazdeljenim pomnilnikom. Nudi nam testni program in program za merjenje časa izvajanja testov.

Benchmark za delovanje potrebuje implementacijo vmesnika sporočilnega sistema (ang. Message Passing Interface - MPI) in implementacijo matematične knjižnice BLAS (ang. Basic Linear Algebra Subprograms) ali VSIPL (the Vector Signal Image Processing Library).

Benchmark HPL smo prenesli s spletne strani [6] in ga raztegnili z ukazom `tar xzf hpl-2.1.tar.gz`. Da bi uspešno zgradili zagonsko datoteko, smo ustvarili tako imenovano Make datoteko. Poimenovana mora biti v obliki `Make.<ARCH>`, kjer lahko `<ARCH>` zamenjamo s poljubnim poimenovanjem. V tej datoteki smo morali navesti vse poti do knjižnic, ki jih HPL potrebuje za delovanje. To smo lahko naredili na dva načina.

(I.) Kopirali smo vzorčno datoteko `Make.Linux_PII_CBLAS` iz direktorija `hpl_intel/setup` in jo poimenovali `Make.hplIntel64` (ARCH = hplIntel64). Popravljene vrstice nastavitev so prikazane na sliki 5.5.

```
ARCH        = hplIntel64
TOPdir      = $(HOME)/hpl_intel
MPdir       = /opt/intel/impi_latest
MPinc       = -I$(MPdir)/include64
MPlib       = $(MPdir)/lib64/libmpi_mt.so
LAdir       = /opt/intel/mkl
LAinc       = -I$(LAdir)/include
LAlib       = -mkl=cluster
CC          = mpiicc
CCNOOPT     = $(HPL_DEFS)
CCFLAGS     = -openmp -xHost -fomit-frame-pointer
              -O3 -funroll-loops $(HPL_DEFS)
LINKER      = $(CC)
```

Slika 5.5: Popravljeni vnosi v nastavitveni datoteki

Make datoteko smo prevedli z ukazom `make arch=hplIntel64`. V primeru, da ni prišlo do napak, se v direktoriju `hpl_intel/bin/hplIntel64` ustvarita datoteki `xhpl` in `HPL.dat`. Z ukazom `ldd xhpl` lahko preverimo, če so vse potrebne dinamične knjižnice najdene. Če je med prevajanjem prišlo do napake, smo z ukazom `make clean_arch_all arch=hplIntel64` pobrisali že prevedene datoteke, popravili make datoteko in ponovno prevedli.

(II.) Kopirali smo datoteke iz Intelove zbirke primerov, ki se nahajajo na poti `/opt/intel/composerxe/mkl/benchmarks/mp_linpack/`, v svoj direktorij in popravili nekaj nastavitev v datoteki `Make.intel64`.

Datoteko lahko prevedemo na tri načine:

1. Ukaz `make arch=intel64` zgradi datoteki `xhpl` in `HPL.dat` v mapi `bin/intel64`.

2. Ukaz `make arch=intel64 version=offload` naredi enako kot 1. ukaz, vendar možnost `offload` določa, da bo CPE delo predala koprocesorju Xeon Phi.

3. Ukaz `ake arch=intel64 version=hybrid` naredi enako kot 1. ukaz, vendar možnost `hybrid` določa, da delo opraviljata CPE in koprocesor Xeon Phi hkrati.

Ne glede na način, ki smo ga uporabili za prevajanje benchmarka (*I.* ali *II.*), uporabimo ustvarjeno datoteko `HPL.dat` za nastavitev zagonskih parametrov benchhmarka. Na sliki 5.6 je podan primer dela datoteke, ki smo ga spremenili. S parametrom `Ns` smo podali velikost problema, s parametrom `NBs` pa velikost blokov, v katere je problem razdeljen. Za parametra `Ps` in `Qs` velja priporočilo, da naj bo razlika med `Ps` in `Qs` čim manjša in v primeru, da nista enaka, naj bo `Qs` večji od `Ps`. Za urejanje te datoteke smo si pomagali z dokumentacijo [6] in s spletnim orodjem na naslovu: `http://www.advancedclustering.com/act-kb/tune-hpl-dat-file/`.

```
8            device out (6=stdout,7=stderr,file)
2            # of problems sizes (N)
35000 78848  Ns
1            # of NBs
128          NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
3            Ps
4            Qs
```

Slika 5.6: Popravljeni vnosi v nastavitveni datoteki

Datoteka `xhpl` je binarna zagonska datoteka, ki jo zaženemo z ukazom `mpirun -n x ./xhpl`, kjer `x` nadomestimo s številom niti, ki jih želimo kreirati ob zagonu. To število mora biti enako produktu parametrov `Ps` in `Qs`. V našem primeru je $x = 12$.

HPL za arhitekturo Intel MIC

Postopek namestitve je zelo podoben prejšnji namestitvi HPL (*II.* postopek). Lahko napišemo svojo datoteko `Make.mic` (ali popravimo vzorčno datoteko) in jo prevedemo z ukazom `make arch=mic`, ali pa uporabimo že prevedene datoteke v direktoriju (`mp_linpack/bin_intel`). Postopek se razlikuje v vsebini datoteke `Make.mic`. Spremembe so prikazane na sliki 5.7. Te spremembe določajo, kje se nahajajo knjižnice in kako jih prevesti.

```
TOPdir      = $(HOME)/mp_linpack
MPdir       = /opt/intel/impi_latest
MPinc       = -I$(MPdir)/mic/include
MPlib       = $(MPdir)/mic/lib/libmpi.a
LAdir       = /opt/intel
MKLINCDIR   = -I/opt/intel/mkl/include
```

```
8         device out
1         # of problems sizes (N)
29696     Ns
1         # of NBs
240       NBs
0         PMAP process mapping
1         # of process grids
1         Ps
1         Qs
```

Slika 5.7: Na levi so prikazane spremembe v datoteki `Make.mic`, na desni pa nastavitve v datoteki `HPL.dat`.

Zaganjanje testa se razlikuje v tem, da ta test zaganjamo v domačem okolju koprocesorja Xeon Phi (gostitelj v tem načinu izvajanja nima nobene vloge). Najprej smo pripravili okolje s kopiranjem potrebnih binarnih datotek in knjižnic, nato pa še z nastavitvijo okoljskih spremenljivk. Nato smo kopirali datoteke benchmarka v domače okolje koprocesorja Xeon Phi. Datoteko `xhpl` zaženemo na enak način kot na gostiteljskem sistemu.

Ker smo do naprave `mic0` (Xeon Phi) želeli dostopati brez gesla, smo si morali nastaviti RSA ključ. Privatni in javni ključ se morata nahajati v domačem direktoriju uporabnika (`$HOME/.ssh`) in biti poimenovana `id_rsa` in `id_rsa.pub`. Ključ smo generirali z ukazom `ssh-keygen`. Dostop brez gesla preko protokola SSH smo nastavili tako, da smo najprej ustavili storitev `mpss` z ukazom `sudo service mpss stop`, nato pa smo z ukazom `sudo micctrl --sshkeys=$HOME` v nastavitve dodali direktorij, kjer so shranjeni naši ključi. V primeru težav z dovoljenji datotek, storitev `mpss` ne mora kopirati podatkov o RSA ključih na napravo `mic0`. Uporabnika smo odstranili z ukazom `micctrl --userdel=user -r` in ga dodali z ukazom `micctrl --useradd=user --uid=535 --gid=504 -c --home=$HOME --sshkeys=$HOME.ssh/ mic0`. Uporabljali smo še ukaze za izbris nastavitev, ponastavitev nastavitev na privzete vrednosti in ukaz za ponovni zagon storitve. Za izvajanje drugih testov smo morali odstraniti tudi koprocesor Xeon Phi. Pred odstranitvijo naprave smo morali preprečiti samodejni zagon storitve `mpss` ob zagonu sistema, saj se v nasprotnem primeru sistem ne zažene. To smo storili z ukazom `sudo chkconfig mpss off`.

HPL za naprave CUDA

Benchmark HPL za okolje CUDA je nekoliko prilagojena različica standardnih testov HPL. Prenesli smo jo z naslova: `https://developer.nvidia.com/rdp/assets/cuda-accelerated-linpack-linux64`, kjer smo se morali najprej registrirati in oddati prošnjo za prijavo na razvojni program *Accelera-*

ted Computing Developer Program. Ko so nam na portalu odobrili prošnjo, smo datoteko prenesli in raztegnili v domači direktorij uporabnika. Datoteko `Make.CUDA` smo popravili (slika 5.8) in prevedli. V direktoriju `/bin/CUDA/` so se kreirale naslednje datoteke: `HPL.dat`, `HPL.dat_example`, `output_example`, `run_linpack` in `xhpl`. V zagonski datoteki `run_linpack` smo uredili podatke o številu jeder procesorja na grafično kartico (teh je 6) in podatek o delitvi dela. Na koncu smo uredili še datoteko `HPL.dat` (slika 5.8). Parametra `Ps` in `Qs` smo določili tako, da je njun produkt enak številu GPE naprav. Teste smo zaganjali z ukazom `mpirun -n 2 ./run_linpack`

```
TOPdir   = $(HOME)/hpl_cuda
MPdir    = /opt/intel/impi_latest
MPinc    = -I$(MPdir)/include64
MPlib    = $(MPdir)/lib64/libmpi.a
LAdir    = /opt/intel/mkl/lib/intel64
LAinc    = -I/usr/local/cuda/include
CC       = mpiicc
CCFLAGS  = $(HPL_DEFS) -O3 -axS -w
    -fomit-frame-pointer -funroll-loops -openmp
```

```
8      device out
1      # of problems sizes (N)
50000  Ns
1      # of NBs
1024   NBs
0      PMAP process mapping
1      # of process grids
1      Ps
2      Qs
```

Slika 5.8: Na levi so prikazane spremembe v datoteki `Make.CUDA`, na desni pa nastavitve v datoteki `HPL.dat`.

5.1.2 The Scalable HeterOgeneous Computing

Zbirka benchmark programov SHOC vsebuje tako OpenCL kot tudi CUDA implementacije. Testi so razdeljeni v tri kategorije. Prva kategorija vsebuje teste z enostavnimi programi za merjenje prenosov in hitrosti. Druga kategorija vsebuje programe z višje-nivojskimi operacijami, kot je hitra fourierova transformacija (FFT) in tretja kategorija vsebuje ščepce zahtevnih aplikacij (na primer ščepec iz simulacije turbulentnega izgorevanja S3D).

Da smo lahko zaganjali teste, smo jih morali najprej prevesti. SHOC vsebuje nastavitveno skripto poimenovano configure, ki smo jo morali popraviti. Odstranili smo vnosa `compute_12` in `compute_13`, saj arhitekturi, ki jih ta

vnosa predstavljata, nista podprti na našem sistemu. Nato smo izvorne datoteke prevedli in namestili z ukazi: `configure --prefix=$PWD/dir`, `make` in `make install`. Vse teste se zažene z ukazom `$PWD/dir/bin/shocdriver -opencl -s 1 -p 0 -d 0,1`, kjer stikalo `-s` predstavlja velikost problemov, stikalo `-p` uporabimo za izbiro platforme in stikalo `-d` za izbiro naprav(e), na katerih želimo teste poganjati. Lahko pa zaženemo vsak test posebej.

5.1.3 Vektorsko seštevanje

Seštevanje je ena najbolj osnovnih aritmetično logičnih operacij. Vektorsko seštevanje smo implementirali v jeziku OpenCL in sešteva dva vektorja dolžine N. Rezultat shranimo v tretji vektor. Podatke za seštevanje smo generirali z generatorjem naključnih števil. Nato smo določili problemsko področje glede na število delovnih skupin, rezervirali pomnilnik na napravi in nanjo prenesli generirane podatke. Začeli smo beležiti čas izvajanja in pognali ščepec. Po izvedbi ščepca smo izračunani vektor prenesli nazaj na gostiteljski sistem.

Vsak delavec, ki izvaja ščepec, je seštel dodeljene vsote in jih shranil na istoležno mesto. V primeru, da problem ni deljiv na cele enake dele, so nekateri delavci opravili eno seštevanje manj.

5.1.4 Množenje vektorjev po komponentah

Množenje dveh vektorjev (a in b) po komponentah

$$c_i = a_i b_i, \tag{5.1}$$

množi dva istoležna elementa v vektorjih in shrani rezultat v istoležno polje vektorja c. Vsi vektorji so dolžine N. Podatke smo prav tako generirali z generatorjem naključnih števil. Določili smo problemsko področje glede na

število delovnih skupin, rezervirali pomnilnik na napravi in nanjo prenesli generirane podatke. Začeli smo meriti čas in pognali ščepec. V primeru, da problem ni deljiv na cele enake dele, so nekateri delavci opravili eno množenje manj.

5.1.5 Skalarni produkt

Skalarni produkt je operacija, ki dvema vektorjema priredi skalar (število). Od množenja vektorjev po komponentah se razlikuje po tem, da zmnožene komponente 5.1 na koncu seštejemo. To prikazuje tudi enačba skalarnega produkta

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n, \tag{5.2}$$

kjer vsak ščepec najprej zmnoži dodeljene elemente. Potem prva nit iz delovne skupine sešteje vse seštevke znotraj skupine in na koncu prva nit iz globalnega prostora sešteje vse seštevke iz delovnih skupin. Paziti smo morali tudi na postavitev sinhronizacijskih točk.

5.1.6 Histogram

Histogram je grafikon, ki prikazuje porazdelitev statistične spremenljivke. Pogosto je uporabljen v statistiki in podatkovnem rudarjenju. Naš program najprej generira `N` naključnih števil med 0 in `M`, nato pa prešteje pojavitve vsakega števila. Ker uporabljamo generator naključnih števil, se pojavitve števil ne razlikujejo veliko.

Poglavje 6

Rezultati

Rezultati meritev so razdeljeni v več delov. Najprej nas je zanimala najvišja poraba sistema v različnih konfiguracijah. Te meritve smo opravljali s stenskim merilnikom Energy Check 3000 proizvajalca Voltcraft. Na koprocesorju Xeon Phi in grafični kartici Tesla pa smo meritve opravljali tudi programsko. Tako smo določili, ali so programske meritve za nas dovolj natančne za nadaljnje primerjave. Nato smo primerjali najvišje porabe benchmark programov HPL (MPI in MKL) in SHOC (OpenCL). Nazadnje smo testirali lastne programe in jim spreminjali argumente, da smo ugotovili, kako vplivajo na samo porabo električne energije.

6.1 Najnižja in najvišja poraba sistema

Ko iščemo podatke o porabi računalniških komponent, najprej prelistamo specifikacije. Večina novih naprav ima navedene podatke o porabi električne energije v stanju mirovanja. Dokaj hitro pridemo tudi do podatkov o TDP (Thermal design power), ki pa se ne nanašajo na porabo električne energije, ampak na toplotno energijo, ki jo proizvede naprava. S podatki o TDP si ne

moremo najbolje pomagati pri določitvi najvišje porabe električne energije, a moramo biti vseeno pozorni nanje. V kolikor hladilni sistem ni zmožen odvesti toplote z naprave, se lahko naprava zaustavi ali pa zmanjša hitrost delovanja. Če sta dve napravi proizvedeni v istem proizvodnem procesu in imata podani oceni TDP, iz tega lahko sklepamo, da ima naprava z višjim TDP tudi višjo porabo električne energije. Na TDP med drugim vpliva frekvenca delovanja, napetosti in tokovi, pri katerih naprava obratuje, hladilni sistem in zunanji pogoji. Najvišjo porabo električne energije grafičnih kartic Tesla in koprocesorja Xeon Phi določajo gonilniki in enota PMU. Preko ukazov `nvidia-smi -a` in `micsmc --freq mic0` smo poizvedeli, kolikšna je omejitev porabe električne energije, preden se kartice začnejo upočasnjevati ali pa se izklopijo. Vrednosti o porabi električne energije v stanju mirovanja (najnižja poraba energije), TDP in dobljene vrednosti o najvišji porabi električne energije so predstavljene v tabeli 6.1.

naprava	poraba v mirovanju `[W]`	TDP `[W]`	najvišja dovoljena poraba `[W]`
Xeon E5-2620	ni podatka	95	ni podatka
Tesla K20m	25	225	225
Xeon Phi 5110P	45	225	306

Tabela 6.1: Podatki o porabi procesorskih enot v stanju mirovanja in TDP iz specifikacij [19, 21, 28] ter najvišja dovoljena poraba, ki jo določata gonilnik in PMU

Najnižjo porabo sistema smo merili v različnih konfiguracijah s stenskim merilnikom. Meritve smo začeli izvajati 5 minut po zagonu sistema, da so se vsi programi zagnali in niso obremenjevali sistema bolj, kot ga ponavadi. Iz tabele 6.2 je razvidno, da grafični kartici v stanju mirovanja porabita toliko energije, kot je zapisano v specifikaciji (skupaj porabita $172W - 122W = 50W$). Koprocesor Xeon Phi se najbolj ne približa podani vrednosti. Po bolj podrobni analizi smo ugotovili, da mora gonilnik za branje podatkov

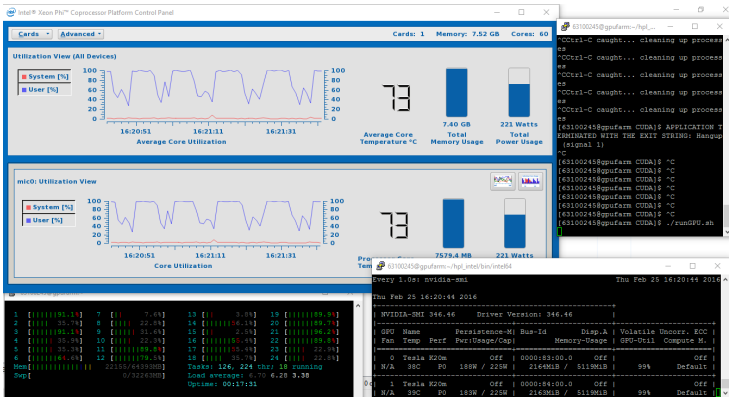
napravo najprej zbuditi iz mirovanja. Tako vsaj eno jedro preide iz mirovanja v izvajalni način C0 (ostali načini so varčevalni) in nato priskrbi podatke o porabi. Zato koprocesor Xeon Phi porabi več energije kot smo najprej pričakovali.

Najvišjo porabo sistema smo prav tako merili v različnih konfiguracijah s stenskim merilnikom. Sistem smo obremenjevali z benchmarkom HPL. Meritve so prikazane v tabeli 6.2. Na prvi pogled imata grafični kartici previsoko porabo za več kot 100 W (razlika med meritvijo in najvišjo dovoljeno porabo). Če dobro pomislimo, pri izvajanju benchmarka sodeluje tudi procesor, ki kot gostitelj izvaja ukaze. Procesor porabi precejšen del energije, da dovolj hitro nadzira delovanje in dostavlja podatke v obdelavo grafičnima karticama. Za koprocesor Xeon Phi smo pričakovali višjo porabo. Po analizi energetskih statusov smo ugotovili, da višjo porabo omejujejo tako imenovana upočasnjevalna stanja. Ker smo presegli TDP naprave, je koprocesor Xeon Phi prešel v upočasnjeno stanje PL0. Z vsakim naslednjim stanjem zmogljivost nekoliko pade, da preprečimo pregrevanje. To pa je nujno, saj tako preprečimo trajne poškodbe naprave. Podrobnosti o upočasnjenih stanjih so opisane v delu [8].

`konfiguracija`	poraba v mirovanju `[W]`	najvišja poraba `[W]`
CPE	122 ± 2	279 ± 2
$2 \times GPE$	$172 - 122 = 50 \pm 4$	$690 - 122 = 568 \pm 4$
$XeonPhi$	$185 - 122 = 63 \pm 4$	$405 - 122 = 283 \pm 4$
$2 \times GPE + XeonPhi$	$245 - 122 = 123 \pm 4$	$924 - 122 = 802 \pm 4$

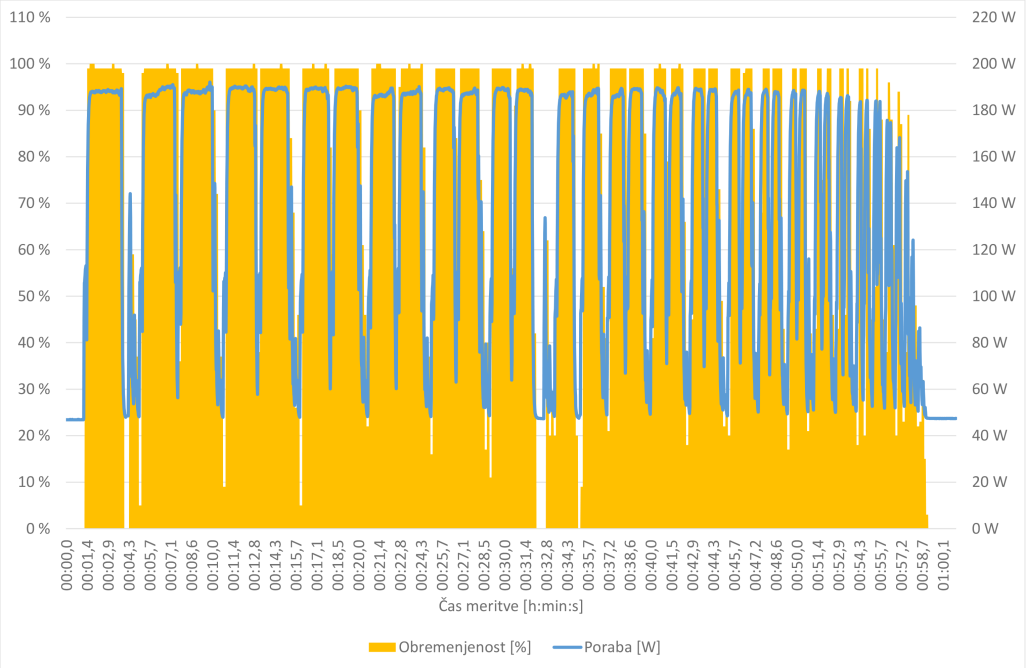
Tabela 6.2: Poraba sistema v različnih konfiguracijah, merjena s stenskim merilnikom. Sistem smo obremenjevali z benchmarkom HPL. Porabo naprav smo izračunali tako, da smo od izmerjene vrednosti odšteli porabo CPE v mirovanju. Sem spadajo tudi napajanje osnovne plošče, diskov in ventilatorjev.

Vzporedno z izvajanjem meritev preko stenskega merilnika smo meritve izvajali tudi programsko (slika 6.1). Na sliki 6.2 je prikazan graf meritev



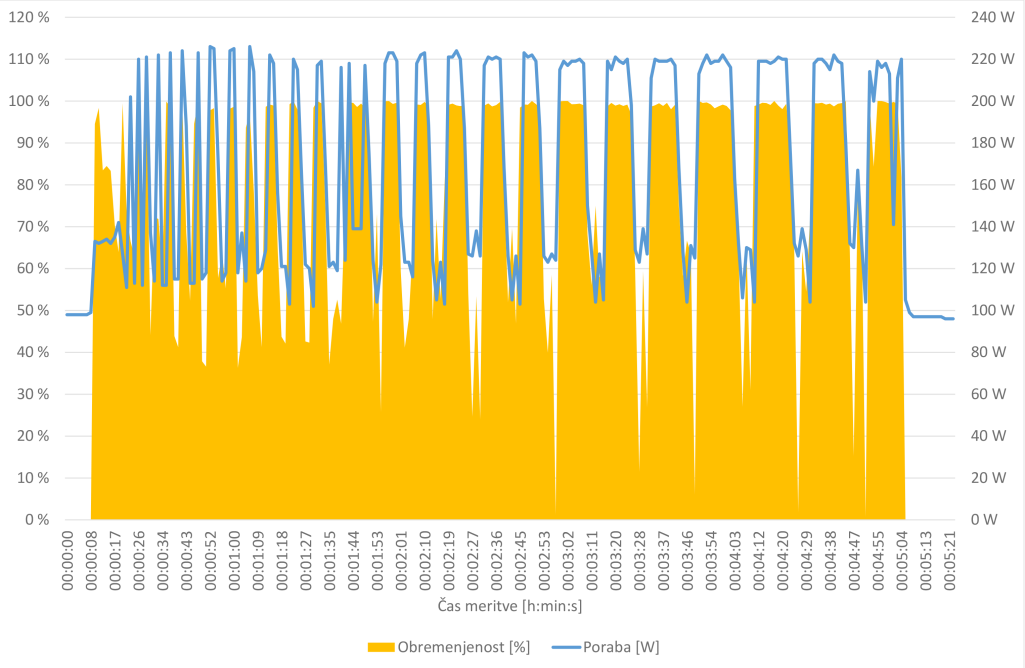
Slika 6.1: Primer izvajanja meritev, kjer preko grafičnega vmesnika spremljamo obremenjenost in porabo koprocesorja. Preko tekstovnega izpisa v konzolo pa spremljamo obremenjenost in porabo energije grafičnih kartic.

prve kartice. Poraba električne energije je tesno povezana z obremenjenostjo GPE. Grafični kartici Tesla sta dosegli najvišjo porabo 197 W in 191 W. Razliko lahko pripišemo absolutni napaki senzorja, kar so opazili tudi M. Burtscher et al. [13]. Njuna skupna najvišja poraba je torej 388 W. Programske meritve so nekoliko nižje od pričakovanih. Ob polni obremenitvi smo pričakovali porabo električne energije med 411 W (izmerjena poraba na merilniku) in 450 W (seštevek najvišjih porab iz specifikacije kartic). Odstopanje v porabi električne energije smo pripisali pretvorbi izmeničnega toka (AC) v enosmernega (DC), saj je izkoristek napajalne enote ravno med 80 % in 90 % (388 W/450 $W = 0,86$ in 388 W/411 $W = 0,94$). Podobne rezultate so opazili tudi S. Kamil et al. [22] in G. Rostirolla et al. [31]. Opazili smo tudi, da med izvajanjem zahtevnih računskih operacij prihaja do neodzivnosti gonilnika in podatkov ne moremo zajemati z željeno frekvenco (na 10 ms ali vsaj 50 ms). Prav tako so nekatere zaporedne meritve niso razlikovale po vrednostih od prejšnjih. Na GPE smo meritve vzorčili na 100 ms, saj smo tako dobili bolj berljive podatke z manj napakami. Podobne anomalije opisujejo tudi M. Burtscher et al. v delu [13].



Slika 6.2: Graf obremenjenosti grafične kartice Tesla in porabe električne energije med izvajanjem benchmarka HPL

Koprocesor Xeon Phi je dosegel najvišjo porabo električne energije 226 W (slika 6.3). Meritev je nižja od pričakovanih 306 W (omejitev nastavljena preko gonilnika). Če dobljeno meritev popravimo zaradi pretvorbe dvosmernega toka v enosmernega, poraba znaša okoli 282 W ($226 \cdot (100/80)$), kar se ujema z meritvami opravljenimi s stenskim merilnikom. Preostala razlika nastane zaradi omejitve TDP in operacijskega sistema Linux, ki ga mora koprocesor Xeon Phi izvajati. Tako naš program ne more povsem izkoristiti računskih jeder in pomnilnika. Na sliki 6.3 vidimo, da se poraba električne energije tesno prilega obremenitvi kartice, razen ob začetku in koncu izvajanja programa. Enota PMU prilagaja porabo električne energije glede na obremenjenost sistema v trenutnem času in ocenjeno porabo v naslednjem. Meritve smo vzorčili na 1 s, saj skoraj toliko traja izvajanje ukaza za pridobivanje podatkov o trenutnem stanju koprocesorja Xeon Phi.



Slika 6.3: Graf obremenjenosti koprocesorja Xeon Phi in porabe električne energije med izvajanjem benchmarka HPL

Pozorni smo bili tudi na druge anomalije, ki so jih opazili avtorji dela [13]. Opazili smo:

- da graf porabe električne energije ni zamaknjen glede na obremenitev jeder (krivulji, ki prikazujeta porabo električne energije na slikah 6.2 in 6.3, se ujemata z obremenitvijo naprav),
- da se prekomerni poskok porabe električne energije ob zagonu programa ne pokaže (predvidevamo, da je izravnava meritev že vgrajena v gonilnik) in
- da poraba električne energije ne pade takoj po koncu izvajanja programa, saj kartica pričakuje, da se bo za trenutnim programom izvedel še kakšen program. V določenih intervalih nato prehaja v bolj varčna stanja delovanja.

6.2 SHOC

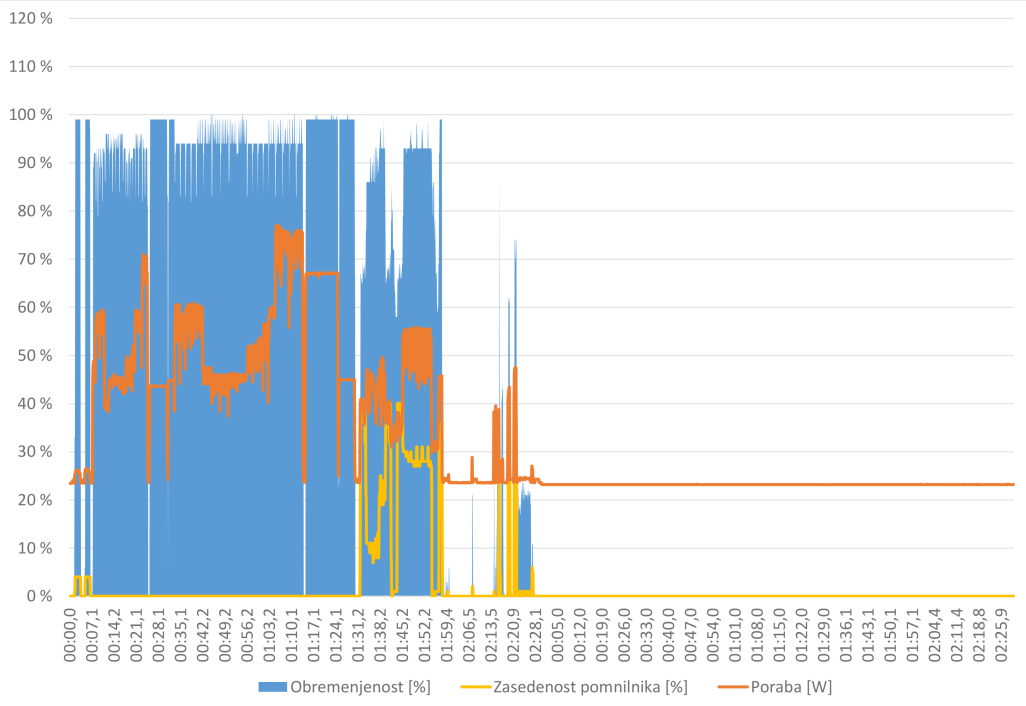
Najvišjo porabo smo merili tudi ob obremenjevanju sistema z benchmarkom SHOC, ki vsebuje implementirane algoritme v programskem jeziku OpenCL. Meritve izmerjene s stenskim merilnikom so predstavljene v tabeli 6.3. Ob

`konfiguracija`	poraba v mirovanju `[W]`	najvišja poraba `[W]`
CPE	122 ± 2	238 ± 2
$2 \times GPE$	$175 - 122 = 53 \pm 4$	$497 - 122 = 375 \pm 4$
$XeonPhi$	$180 - 122 = 58 \pm 4$	$332 - 122 = 210 \pm 4$
$2 \times GPE + XeonPhi$	$248 - 122 = 126 \pm 4$	$685 - 122 = 563 \pm 4$

Tabela 6.3: Poraba sistema v različnih konfiguracijah, merjena s stenskim merilnikom, ob obremenjevanju sistema z benchmarkom SHOC. Porabo naprav smo izračunali tako, da smo od izmerjene vrednosti odšteli porabo CPE v mirovanju. Sem spada tudi napajanje osnovne plošče, diskov in ventilatorjev.

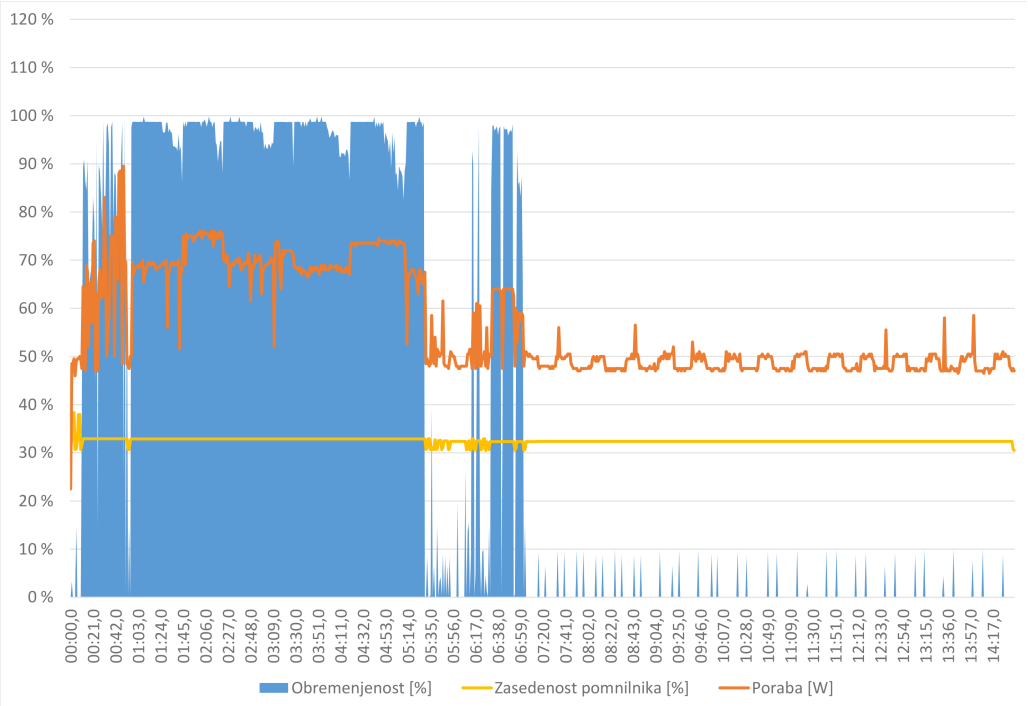
primerjavi porabe v mirovanju z vrednostmi iz tabele 6.2 vidimo, da porabe nihajo znotraj absolutne napake merilnika. Najvišja poraba ob obremenjevanju sistema z benchmarkom SHOC je bila nekoliko nižja v primerjavi z obremenjevanjem z benchmarkom HPL. Opazili smo tudi, da je poraba grafičnih kartic veliko nižja. Ko je gostitelj predal delo grafičnima karticama Tesla, mu ni bilo potrebno tako tesno sodelovanje z napravama, kot v primeru HPL. Tako CPE porabi veliko manj energije. Kljub temu, da je najvišja poraba sistema obremenjenega z benchmarkom SHOC nižja, pa ne smemo sklepati na boljšo učinkovitost. Benchmarka HPL in SHOC se namreč razlikujeta po vključenih testih in posledično testnem bremenu.

Programske meritve, opravljene na grafični kartici Tesla in na koprocesorju Xeon Phi, so prikazane na slikah 6.4 in 6.5. Če primerjamo obliko



Slika 6.4: Obremenjenost grafične kartice Tesla, zasedenost njenega pomnilnika in poraba električne energije med izvajanjem benchmarka SHOC

grafov, opazimo, da sta si podobni. Največja razlika je v trajanju izvajanja istega testnega bremena in porabi pomnilnika. Grafična kartica Tesla je izvedla benchmark v 2 min in 30 s, medtem ko je koprocesor Xeon Phi potreboval kar 14 min in 37 s. Temu primerna je tudi poraba električne energije. Grafična kartica Tesla je porabila 3,7 W, koprocesor Xeon Phi pa kar 27 W. Glavna razlika je v nasploh višji začetni porabi koprocesorja Xeon Phi. Če primerjamo programske meritve z meritvami opravljenimi s stenskim merilnikom, se nahajajo v območju učinkovitosti napajalnika, zato smo nadaljnje meritve opravljali samo programsko.



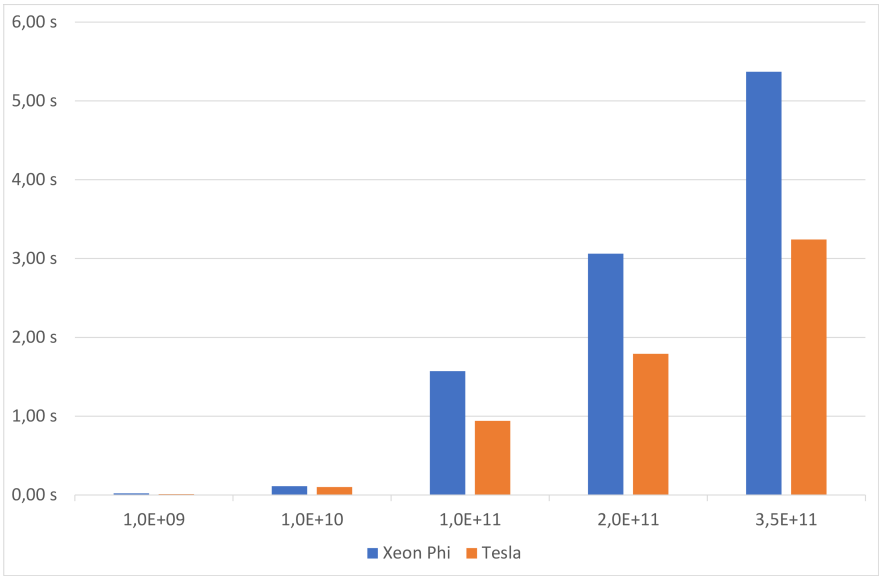
Slika 6.5: Obremenjenost koprocesorja Xeon Phi, zasedenost njegovega pomnilnika in poraba električne energije med izvajanjem benchmarka SHOC

6.3 Poraba energije glede na izbrani algoritem in parametre

Vektorsko seštevanje je zelo hitra operacija, ki porabi malo energije. Merili smo, kako število elementov in velikost delovnih skupin vplivata na porabo električne energije. Rezultati so predstavljeni v tabeli 6.4 in na sliki 6.6. Povečevanje števila podatkov sorazmerno poveča čas izvajanja programa.

N	10^9	10^{10}	10^{11}	$2 \cdot 10^{11}$	$3,5 \cdot 10^{11}$
$t_{Tesla}[s]$	$0,01 \pm 0,01$	$0,10 \pm 0,01$	$0,94 \pm 0,1$	$1,79 \pm 0,1$	$3,24 \pm 0,1$
$t_{XeonPhi}[s]$	$0,02 \pm 0,01$	$0,11 \pm 0,01$	$1,57 \pm 0,1$	$3,06 \pm 0,1$	$5,37 \pm 0,1$

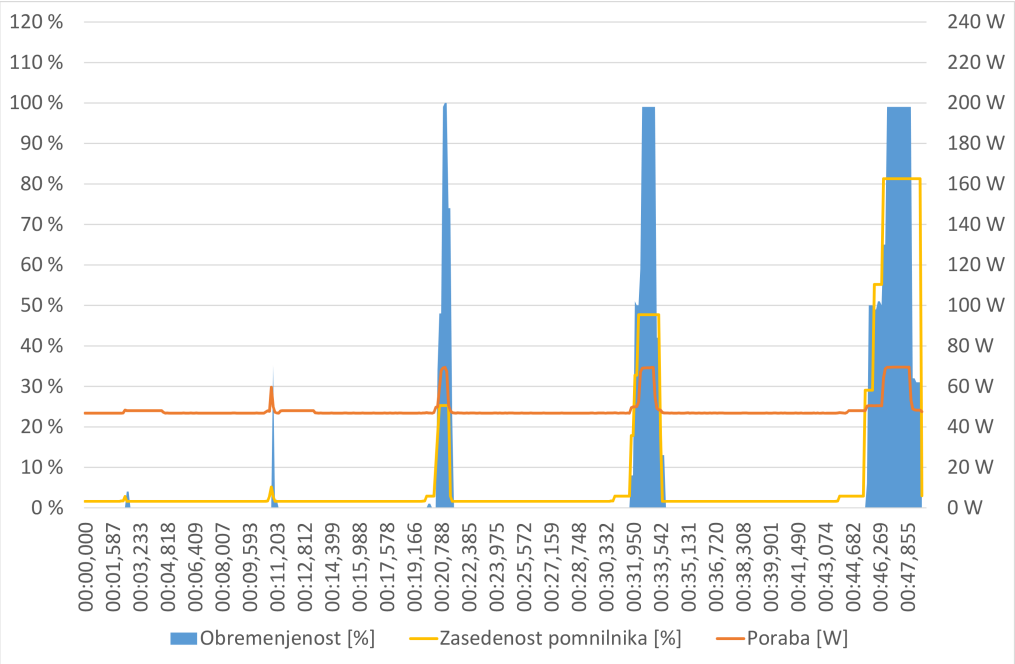
Tabela 6.4: Čas t izvajanja seštevanja glede na število elementov N



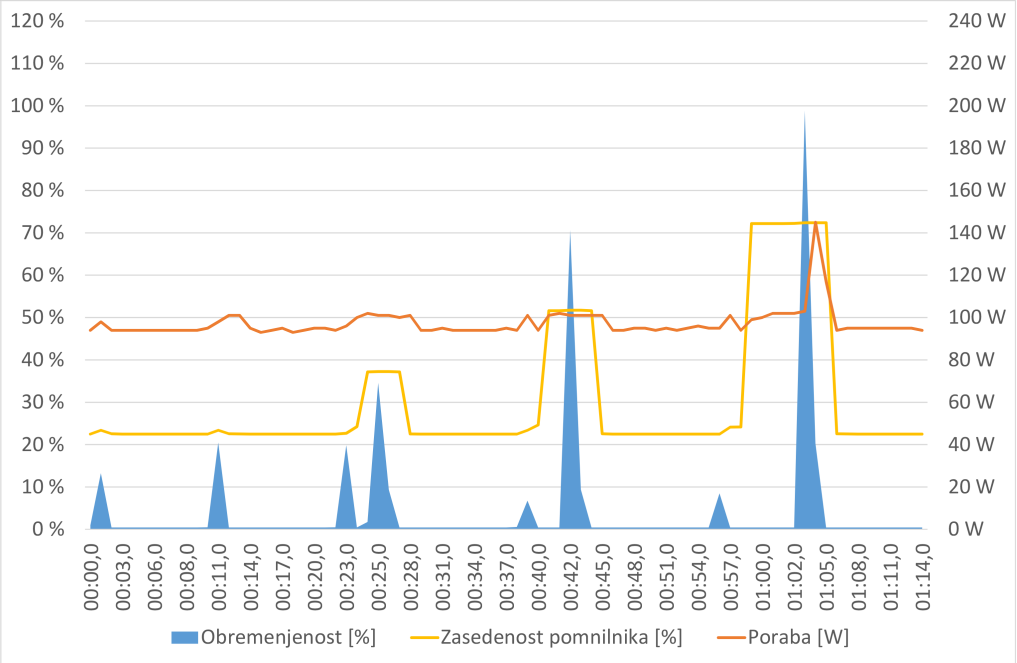
Slika 6.6: Čas izvajanja seštevanja N elementov na koprocesorju Xeon Phi in grafični kartici Tesla

Opazili smo, da je grafična kartica Tesla dvakrat hitreje izvedla seštevanja. Tak rezultat smo pričakovali, saj lahko izvede več operacij na sekundo. Graf

porabe električne energije tesno sovpada z obremenjenostjo jeder grafične kartice Tesla (slika 6.7) in koprocesorja Xeon Phi (slika 6.8). Iz grafa na sliki 6.8 je razvidno, da je koprocesor Xeon Phi veliko procesorske moči in časa porabil za prenos podatkov na napravo in iz nje, saj ima počasnejše podatkovno vodilo kot grafična kartica Tesla. Za samo izvajanje računskih operacij ne potrebuje veliko procesorske moči. To mu omogočajo 512 bitne vektorske procesne enote.

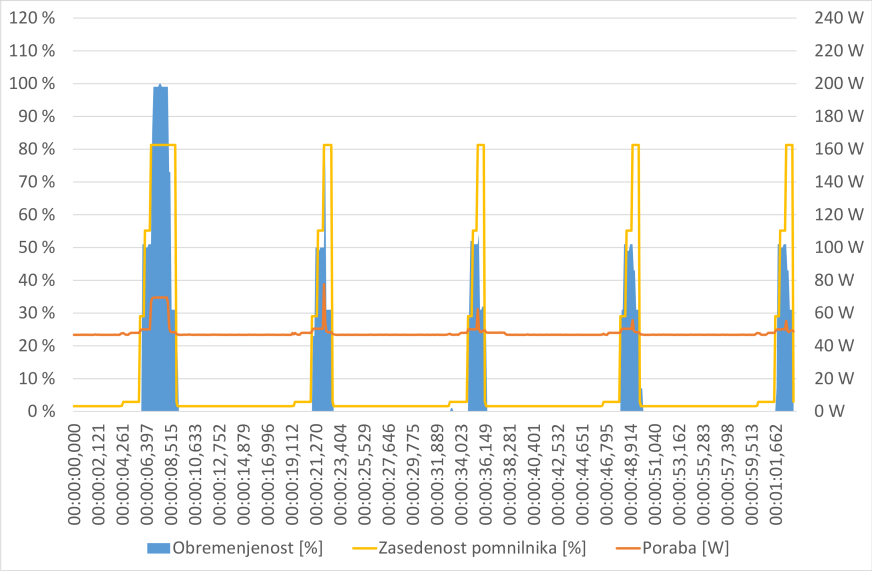


Slika 6.7: Obremenjenost grafične kartice Tesla ob izvajanju vektorskega seštevanja

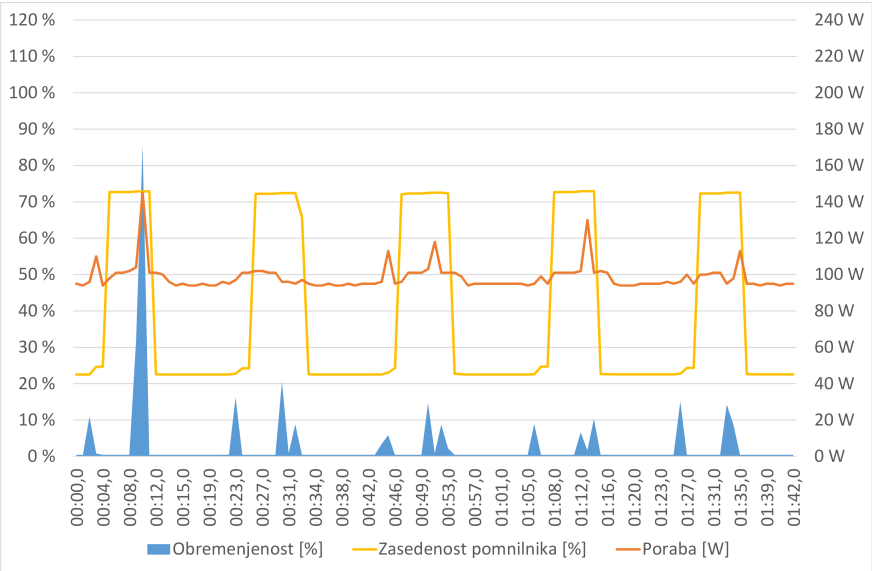


Slika 6.8: Obremenjenost koprocesorja Xeon Phi ob izvajanju vektorskega seštevanja

Zanimalo nas je tudi, koliko na izvajanje vpliva izbira velikosti delovne skupine (sliki 6.9 in 6.10). Teste smo izvajali nad $3,5 \cdot 10^{11}$ števili. S tem smo zapolnili skoraj ves pomnilnik obeh naprav. Velikosti delovnih skupin so bile: 1, 16, 64, 256 in 1024. Na grafični kartici Tesla je vidna pohitritev ob uporabi delovne skupine večje od 1. Vse naslednje meritve so si zelo podobne, saj testno breme ni dovolj zahtevno in optimizirano za vzporedno izvajanje. Na koprocesorju Xeon Phi
pohitritev v izvajanju nismo opazili.



Slika 6.9: Obremenjenost grafične kartice Tesla ob izvajanju vektorskega seštevanja nad $3,5 \cdot 10^{11}$ podatki in različnim številom niti v delovni skupini



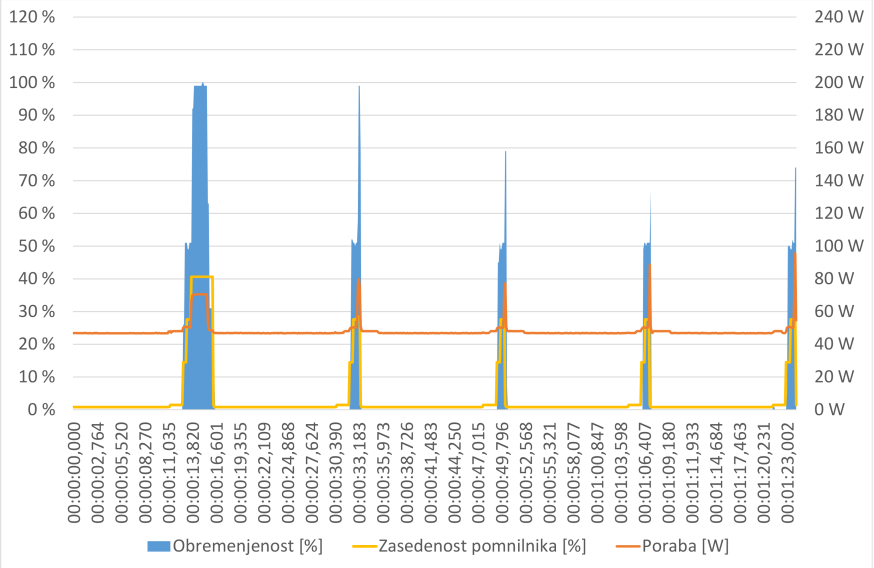
Slika 6.10: Obremenjenost koprocesorja Xeon Phi ob izvajanju vektorskega seštevanja nad $3,5 \cdot 10^{11}$ podatki in različnim številom niti v delovni skupini

Meritve množenja vektorjev po komponentah so se izkazale kot nepotrebne, saj smo ugotovili, da je samo izvajanje, čas izvajanja in poraba električne energije zelo podobno izvajanju seštevanja.

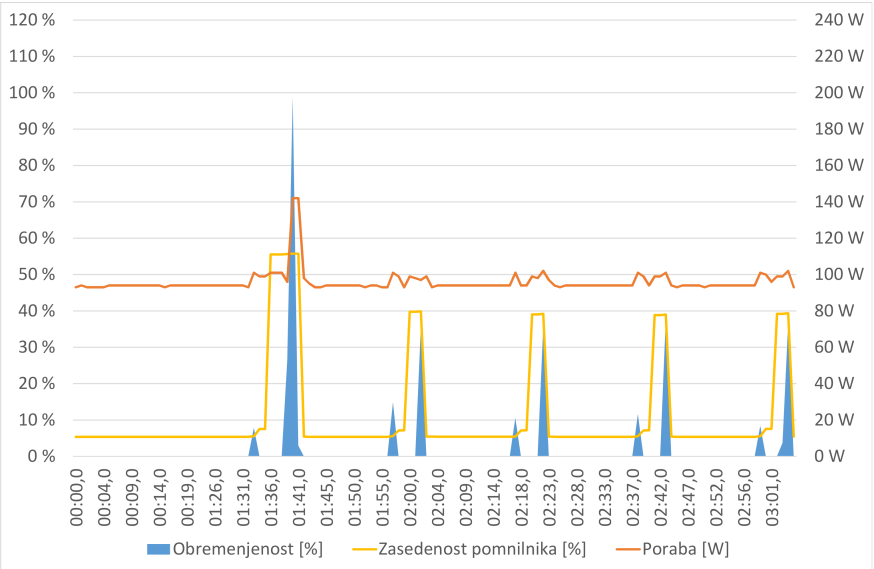
Skalarni produkt se z velikostjo delovne skupine 1 izvaja zelo podobno kot množenje vektorjev po komponentah in seštevanje, zato meritev na različnem številu podatkov nismo opravljali. Zanimale pa so nas meritve z različno velikimi delovnimi skupinami. Na slikah 6.11 in 6.12 je prikazano izvajanje skalarnega produkta nad vektorji velikosti $3,5 \cdot 10^{11}$. Meritve smo izvajali nad delovnimi skupinami različnih velikosti (glej tabelo 6.5). Poraba električne energije se tudi tukaj tesno prilega obremenjenosti računskih naprav.Pri meritvah algoritma na obeh karticah z eno delovno skupino je opazno, da je poraba električne energije višja kot pri ostalih velikostih skupin.

N	1	16	64	256	1024
$t_{Tesla}[s]$	$3,37 \pm 0,1$	$1,24 \pm 0,1$	$0,97 \pm 0,1$	$0,96 \pm 0,1$	$0,99 \pm 0,1$
$t_{XeonPhi}[s]$	$5,58 \pm 0,1$	$2,87 \pm 0,1$	$2,73 \pm 0,1$	$2,73 \pm 0,1$	$2,80 \pm 0,1$

Tabela 6.5: Čas t izvajanja seštevanja glede na število elementov N

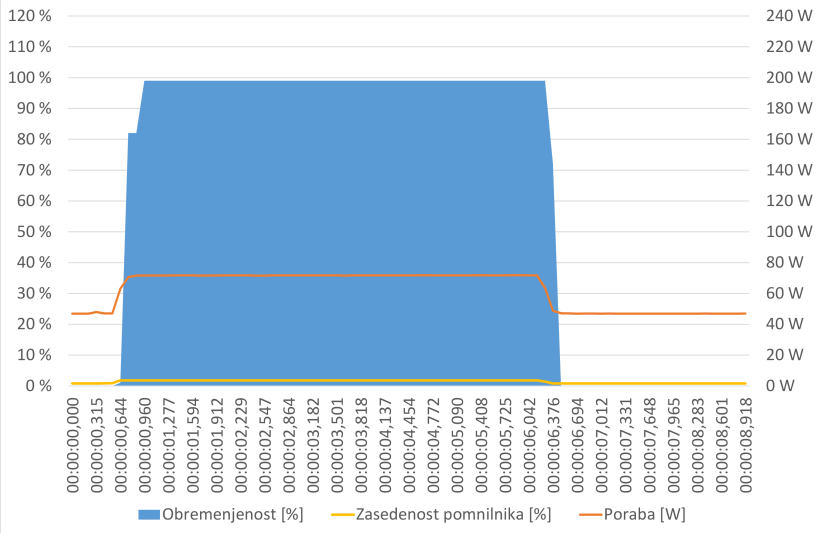


Slika 6.11: Obremenjenost grafične kartice Tesla ob izvajanju skalarnega produkta nad vektorji s $3,5 \cdot 10^{11}$ podatki in različnim številom niti v delovni skupini



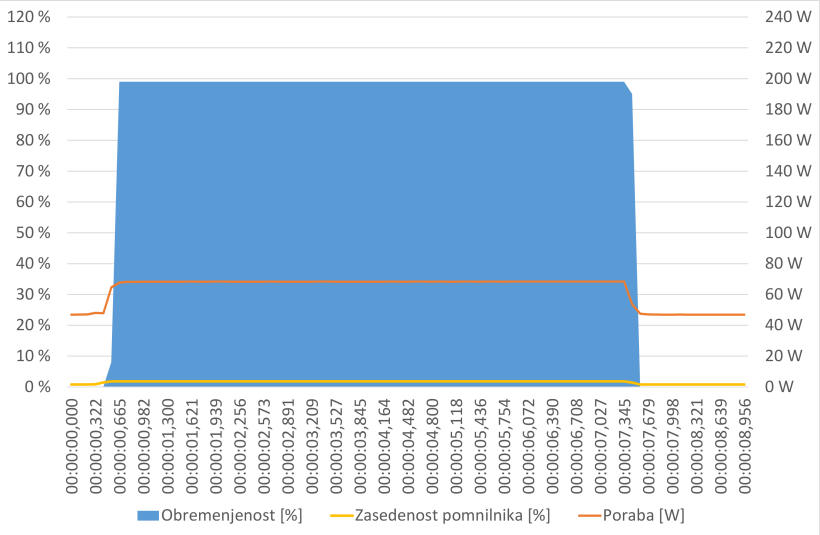
Slika 6.12: Obremenjenost koprocesorja Xeon Phi ob izvajanju skalarnega produkta nad vektorji s $3,5 \cdot 10^{11}$ podatki in različnim številom niti v delovni skupini

S pomočjo algoritma za izračunavanje histograma smo testirali, kako na porabo in čas izvajanja vpliva uporaba lokalnega pomnilnika. Na sliki 6.13 je prikazan graf izračunavanja histograma z globalnim pomnilnikom. Jedra smo obremenili do 99 %, najvišja poraba pa je bila 73 W. Na sliki 6.14 je prikazan graf izvajanja algoritma z lokalnim pomnilnikom. Ta ščepec se je izvajal 0,1 s dlje od različice brez lokalnega pomnilnika. Najvišja poraba je bila 68 W, porabil pa je tudi manj električne energije. Program brez lokalnega pomnilnika je porabil 6,58 mW, program z lokalnim pomnilnikom je porabil 6,47 mW. Če bi moral ščepec brez lokalnega pomnilnika izvesti več operacij nad istimi podatki, bi bila poraba glede na ščepec z lokalnim pomnilnikom še višja.



Slika 6.13: Obremenjenost grafične kartice Tesla ob izračunavanju histograma

Zanimalo nas je tudi, kako na porabo električne energije vpliva število zagnanih ščepcev. Testirali smo dva primera. V prvem smo zaganjali 100 ščepcev enega za drugim in preračunavali vektorje velikosti $200 \cdot 10^9$. Ščepci so se izvajali 24,2 s, porabili pa so 19,633 mW. V drugem primeru smo zaganjali 800 ščepcev in preračunavali vektorje velikosti $25 \cdot 10^9$. Ščepci so se izvajali 23,6 s, porabili pa so 19,298 mW. V obeh primerih smo dosegli



Slika 6.14: Obremenjenost grafične kartice Tesla ob izračunavanju histograma s pomočjo lokalnega pomnilnika

najvišjo porabo 72 W, čas izvajanja pa se razlikuje za 0,6 s. Razlika časov je dovolj majhna, da lahko za naš ščepec trdimo, da število zagnanih ščepcev bistveno ne vpliva na čas izvajanja. Nastala razlika v porabi pa nastane zaradi lokalnosti dostopov do pomnilnika. Za določitev optimalnega števila zagonov ščepcev bi bila potrebna nadaljnja testiranja.

Poglavje 7

Sklepne ugotovitve

V diplomski nalogi smo predstavili programsko in strojno opremo, ki se uporablja za opravljanje meritev porabe električne energije računalniških sistemov. Ker poraba električne energije predstavlja oviro v nadaljnjem razvoju večjih sistemov, nas je zanimalo, če lahko programer na kakršenkoli način vpliva nanjo. Odločili smo se, da izmerimo porabo električne energije nekaterih programov napisanih v programskem jeziku OpenCL. Ugotavljali smo, kako različni parametri programskega jezika OpenCL vplivajo na porabo električne energije.

Naprave za merjenje porabe električne energije se v grobem delijo na samostoječe (na primer stenski merilniki in tokovne klešče) in vgrajene rešitve (integrirani senzorji v vezja in same porabnike). Opisali smo slabosti in prednosti različnih metod. Ugotovili smo, da lahko meritve večjih sistemov izvajamo na manjšem podsistemu in pridobljene podatke posplošimo. Za manjši podsistem lahko vzamemo tudi naš strežnik z dvema procesorjema Intel Xeon E5-2620, 64 GB sistemskega pomnilnika, dvema grafičnima karticama Nvidia Tesla K20m in koprocesorjem Intel Xeon Phi 5110P. Testiranje na podsistemu je marsikje edina možnost, saj težko najdemo primeren čas,

da sistem izklopimo. Lahko pa se pojavijo tudi druge težave, kot na primer nedostopnost sistema zaradi nepravilne konfiguracije sistema.

Meritve različnih bremen smo izvajali s stenskim merilnikom in programsko. Ugotovili smo, da so programske meritve nižje od meritev pridobljenih s stenskim merilnikom, saj napajalnik, ki pretvarja izmenični tok v enosmernega, ni 100 % učinkovit. Naš napajalnik ima ocenjeno učinkovitost med 80 % in 90 %, izračunani izkoristek pa se nahaja med 86 % in 94 %. Če programske meritve delimo z izkoristkom, dobimo realno porabo naprave. Meritev, pridobljenih s stenskim merilnikom, prav tako ne moremo učinkovito zbirati in podatkov vizualizirati. Med izvajanjem programskih meritev z visoko frekvenco vzorčenja smo naleteli na neodzivnost gonilnika in nekatere zaporedne meritve so ostajale enake. Zato smo se odločili za vzorčenje z nižjo frekvenco.

Na porabo električne energije najbolj vpliva količina vhodnih podatkov, raba strojne opreme in sam čas izvajanja programa. Koprocesor Xeon Phi ima ob lažjem bremenu višjo porabo kot grafična kartica Tesla. Da se začetna razlika porabe izniči, moramo kartice zelo obremeniti. Z našimi programi nam to ni najbolje uspelo. Če je program optimiziran za hitro izvajanje, bo ponavadi tudi poraba nižja. Na čas izvajanja smo vplivali tudi s številom podatkov, ki smo jih morali kopirati na kartice in iz njih.

Zelo pomembna je tudi izbira velikosti in števila delovnih skupin. Od tega je odvisna stopnja paralelnosti našega programa in tudi hitrost izvajanja. Če je skupin malo, bodo določena jedra ostala neobremenjena, druga pa bodo garala. Zanimalo nas je še, koliko na hitrost vpliva uporaba lokalnega pomnilnika in število zagnanih jeder. Naši programi so se izkazali kot neprimerni za tovrstna testiranja, saj so preveč monotoni.

Literatura

[1] Alegro. [Online]. Dosegljivo: http://www.allegromicro.com/en/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/ACS712.aspx. [Dostopano 1. 3. 2016].

[2] Ampermetri s kleščami. [Online]. Dosegljivo: http://www.voltcraft.com/sl/ampermetri-s-klescami/. [Dostopano 1. 3. 2016].

[3] Dennard scaling. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Dennard_scaling. [Dostopano 1. 3. 2016].

[4] Determining the idle power of an Intel® Xeon Phi™ coprocessor. [Online]. Dosegljivo: https://software.intel.com/en-us/articles/determining-the-idle-power-of-an-intel-xeon-phi-coprocessor. [Dostopano 1. 3. 2016].

[5] General-purpose computing on graphics processing units. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units. [Dostopano 1. 3. 2016].

[6] Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. [Online]. Dosegljivo: http://www.netlib.org/benchmark/hpl/. [Dostopano 1. 3. 2016].

[7] Intel® Xeon Phi™ coprocessor power management configuration: Using the micsmc command-line interface. [Online]. Dosegljivo:

https://software.intel.com/en-us/blogs/2014/01/31/intel-xeon-phi-coprocessor-power-management-configuration-using-the-micsmc-command. [Dostopano 1. 3. 2016].

[8] Intel® Xeon Phi™ coprocessor x100 product family. [Online]. Dosegljivo: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html. [Dostopano 9. 2. 2017].

[9] OpenCL. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/OpenCL. [Dostopano 1. 3. 2016].

[10] The shoc benchmark suite. [Online]. Dosegljivo: https://github.com/vetter/shoc. [Dostopano 1. 3. 2016].

[11] N. Bates, C. H. Hsu, N. Imam, T. Wilde, and D. Sartor. Re-examining hpc energy efficiency dashboard elements. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1106–1109, May 2016.

[12] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[13] Martin Burtscher, Ivan Zecena, and Ziliang Zong. Measuring gpu power with the K20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 28. ACM, 2014.

[14] Xi Chen, Zheng Xu, Hyungjun Kim, Paul V Gratz, Jiang Hu, Michael Kishinevsky, Umit Ogras, and Raid Ayoub. Dynamic voltage and frequency scaling for shared resources in multicore processor designs. In *Proceedings of the 50th Annual Design Automation Conference*, page 114. ACM, 2013.

[15] CPU-Z. Cpu-z oc world records. [Online]. Dosegljivo: http://valid.canardpc.com/records.php. [Dostopano 1. 3. 2016].

[16] Dravske elektrarne Maribor d.o.o. Splošni podatki o proizvodnji: tokokrog zanesljivega pridobivanja električne energije. [Online]. Dosegljivo: http://www.dem.si/sl-si/Elektrarne-in-proizvodnja/Splo%C5%A1ni-podatki. [Dostopano 1. 3. 2016].

[17] Jason Flinn and Mahadev Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[18] Green500. The Green500 list - november 2015. [Online]. Dosegljivo: http://www.green500.org/lists/green201511. [Dostopano 1. 3. 2016].

[19] Intel. Intel® Many Integrated Core Architecture. [Online]. Dosegljivo: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. [Dostopano 1. 3. 2016].

[20] Intel. Intel® Xeon Phi™ Coprocessor 5110P. [Online]. Dosegljivo: http://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core. [Dostopano 1. 3. 2016].

[21] Intel. Intel® Xeon® Processor E5-2620. [Online]. Dosegljivo: http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI. [Dostopano 1. 3. 2016].

[22] Shoaib Kamil, John Shalf, and Erich Strohmaier. Power efficiency in high performance computing. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[23] Khronos OpenCL Working Group. The OpenCL specification. [Online]. Dosegljivo: `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`. [Dostopano 1. 3. 2016].

[24] Dušan Kodek. *Arhitektura in organizacija računalniških sistemov*. Bitim, 2008.

[25] James H Laros III, Kevin T Pedretti, Suzanne M Kelly, John P Vandyke, Kurt B Ferreira, Courtenay T Vaughan, and Mark Swan. Topics on measuring real power usage on high performance computing platforms. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[26] James Howard Laros III. *Measuring and tuning energy efficiency on large scale high performance computing platforms*. PhD thesis, 2012.

[27] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[28] Nvidia. Tesla K20 GPU accelerator. [Online]. Dosegljivo: `http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf`. [Dostopano 1. 3. 2016].

[29] Edson Luiz Padoin, Philippe OA Navaux, and Grupo de Processamento Paralelo e Distribuido. HPC vs HPC High Performance Computing vs High Power Consumption. 2012.

[30] Rezaur Rahman. *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[31] Gustavo Rostirolla, Rodrigo da Rosa Righi, Vinicius Facco Rodrigues, Pedro Velho, and Edson Luiz Padoin. GreenHPC: a novel framework to measure energy consumption on HPC applications. *Sustainable Internet and ICT for Sustainability (SustainIT), 2015*, pages 1–8, 2015.

[32] SPEC. SPEC power committee. [Online]. Dosegljivo: https://www.spec.org/power/. [Dostopano 1. 3. 2016].

[33] Balaji Subramaniam and Wu-chun Feng. Statistical power and performance modeling for optimizing the energy efficiency of scientific computing. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 139–146. IEEE, 2010.

[34] Supermicro. Supermicro X9DRG-QF. [Online]. Dosegljivo: http://www.supermicro.com/products/motherboard/Xeon/C600/X9DRG-QF.cfm. [Dostopano 1. 3. 2016].

[35] The Khronos Group. Opencl. [Online]. Dosegljivo: https://www.khronos.org/opencl/. [Dostopano 1. 3. 2016].

[36] Top500. Top500 list - november 2015. [Online]. Dosegljivo: http://www.top500.org/lists/2015/11/. [Dostopano 1. 3. 2016].

[37] Yash Ukidave, Amir Kavyan Ziabari, Perhaad Mistry, Gunar Schirner, and David Kaeli. Quantifying the energy efficiency of fft on heterogeneous platforms. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 235–244. IEEE, 2013.